ColorSync 2.0 • Source Control With SourceServer • About Scripting Dictionaries

develop

The Apple Technical Journal

Issue 23 September 1995

Music the Easy Way: The QuickTime Music Architecture

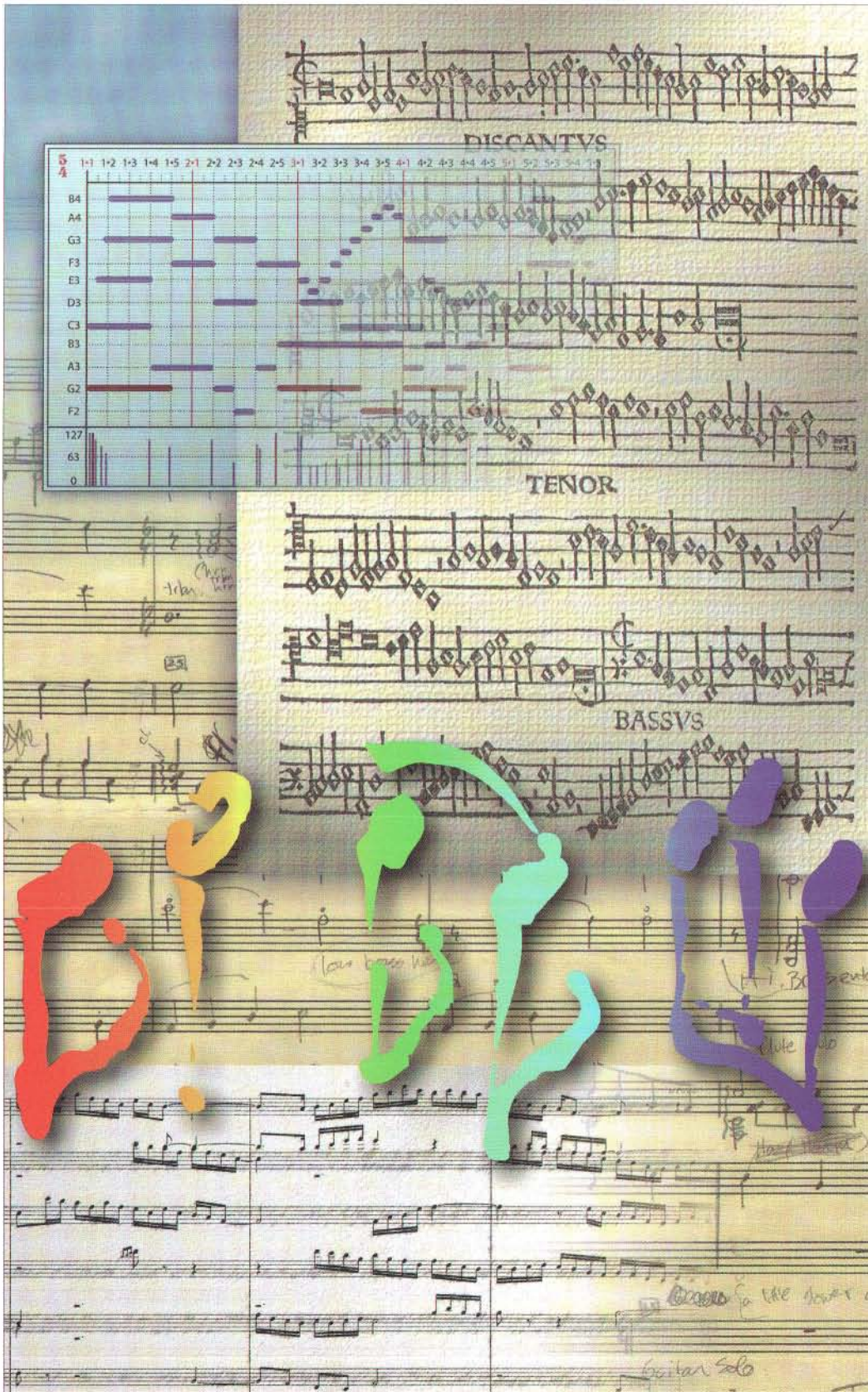
The Basics of QuickDraw 3D Geometries

Implementing Shared Internet Preferences With Internet Config

Multipane Dialogs

Document Synchronization

Power Macintosh: The Next Generation



$10.00

develop

EDITORIAL STAFF

Editor-in-Cheek *Caroline Rose*

Managing Editor *Toni Moccia*

Technical Buckstopper *Dave Johnson*

Bookmark CD Leader *Alex Dosher*

Able Assistant *Meredith Best*

Our Boss *Mark Bloomquist*

His Boss *Dennis Matthews*

Review Board *Jim Luther, Dave Radcliffe, Jim Reekes, Bryan K. "Beaker" Ressler, Larry Rosenstein, Andy Shebanow, Gregg Williams*

Contributing Editors *Lorraine Anderson, Toni Haskell, Judy Helfand, Tim Monroe, Cheryl Potter, Joan Stigliani*

Indexer *Marc Savage*

ART & PRODUCTION

Production Manager *Diane Wilcox*

Technical Illustration *Mary Prusmack Ching, Deb Dennis, John Ryan, Laurie Wigham*

Formatting *Forbes Mill Press*

Photography *Sharon Beals, Maggie Fishell, Gretchen Linton*

Cover Illustration *Graham Metcalfe of Metcalfe/Shubert Design*

ISSN #1047-0735.

THINGS TO KNOW

develop, The Apple Technical Journal, a quarterly publication of Apple Computer's Developer Press group, is published in March, June, September, and December. *develop* articles and code have been reviewed for robustness by Apple engineers.

This issue's CD. Subscription issues of *develop* are accompanied by the *develop Bookmark* CD. This CD contains a subset of the materials on the monthly *Developer CD Series*, available from APDA. Included on the CD are this issue and all back issues of *develop* along with the code that the articles describe. (The code is updated periodically, so always use the most recent CD.) The CD also contains Technical Notes, sample code, and other documentation and tools (these contents are subject to change). Items referred to as being on "this issue's CD" are located on either the Bookmark CD or the Reference Library or Tool Chest edition of the *Developer CD Series*. The *develop* issues and code are also available in the Developer Services areas on AppleLink and eWorld and at ftp.info.apple.com. (Selected articles are on the World Wide Web at http://www.apple.com, in the Developer Services area.)

Macintosh Technical Notes. Where references to Macintosh Technical Notes in *develop* are followed by something like "(QT 4)," this indicates the category and number of the Note on this issue's CD. (QT is the QuickTime category.)

E-mail addresses. Most e-mail addresses mentioned in *develop* are AppleLink addresses; to convert one of these to an Internet address, append "@applelink.apple.com" to it. For example, DEVELOP on AppleLink becomes develop@applelink.apple.com on the Internet. Append "@eworld.com" to eWorld addresses, and append "@online.apple.com" to NewtonMail addresses.

CONTACTING US

Feedback. Send editorial suggestions or comments to Caroline Rose at AppleLink CROSE, Internet crose@applelink.apple.com, or fax (408)974-6395. Send technical questions about *develop* to Dave Johnson at AppleLink JOHNSON.DK, Internet dkj@apple.com, CompuServe 75300,715, or fax (408)974-6395. Or write to Caroline or Dave at Apple Computer, Inc., 1 Infinite Loop, M/S 303-4DP, Cupertino, CA 95014.

Article submissions. Ask for our Author's Guidelines and a submission form at AppleLink DEVELOP, Internet develop@applelink.apple.com, or fax (408)974-6395. Or write to Caroline Rose at the above address.

Subscriptions and back issues. You can subscribe to *develop* through APDA (see ordering information below) or use the subscription card in this issue. You can also order printed back issues from APDA. For all subscription changes or queries, contact APDA and *be sure to include your name, address, and account number as it appears on your mailing label.*

The one-year U.S. subscription price is $30 (for 4 issues and 4 *develop Bookmark* CDs), or $50 U.S. in other countries. Back issues are $13 each. These prices include shipping and handling. For Canadian orders, the subscription price includes GST (R100236199).

APDA. To order products from APDA or receive the *Apple Developer Tools Catalog* of all the products available from APDA, call 1-800-282-2732 in the U.S., 1-800-637-0029 in Canada, (716)871-6555 internationally, or (716)871-6511 for fax. Order electronically at AppleLink APDA, Internet apda@applelink.apple.com, CompuServe 76666,2405, or America Online APDAorder. Or write APDA, Apple Computer, Inc., P.O. Box 319, Buffalo, NY 14207-0319.

Issue 23 September 1995

ARTICLES

COLUMNS

EDITOR'S NOTE

CAROLINE ROSE

On this issue's CD, all the files that used to be in Apple DocViewer format have been converted to Adobe™ Acrobat™. Based on feedback that we've gotten from many of you since we started using DocViewer, we trust you'll be happy with this change. You should find that Acrobat has better search features and resolves some other problems. Because conversion is faster, the information can be more timely. Also, the files take up less space. So we hope you're satisfied — but you probably won't be for very long. It's just not the nature of the computer-using beast.

Think about it: How long after you get an upgrade to some software or hardware product do you start looking ahead to the next version? With the old problems solved and your old needs satisfied, you go on to realize a set of new ones. When it comes to computers, we always want more, and better.

I remember when the Macintosh was first designed, Steve Jobs kept saying it was to be an appliance, like a toaster: you simply plug it in and it does what you want, reliably and without fuss. (We're not talking multi-attachment Cuisinart here.) As you no doubt recall only too well, the options to add to the functionality of the first Macintosh were intentionally limited. It was to be a simple "black box" ("beige box"?).

Now, I know you're glad that that era didn't last very long, but think for a moment about all those non-nerds out there who haven't yet seen fit to buy a computer — all those potential customers Jobs was hoping to attract. They write things and add up numbers sometimes just like we do, don't they? So what are they waiting for?

I think the problem is that they want toasters — machines that work year after year without always needing to be updated, upgraded, or extended. They see the computer as a moving target, constantly advancing to satisfy some relatively insignificant new needs they never even knew they had — doomed to instant obsolescence. Better they should use a pen or pencil.

I suffer from this attitude myself to some degree, at least when I'm wearing my Home User hat. There I use a little old Macintosh with old but reliable software that works every time I do the paperwork on it that I've been doing for ten years now. But at Apple, I become a Computer Professional monster with ravenous needs for the latest and greatest software and hardware — downright insatiable.

So enjoy it while you can: have your fill of our new Acrobat files or whatever innovation pleases you these days. But rest assured that you'll be hungry again in no time.

Caroline Rose

Caroline Rose
Editor

CAROLINE ROSE (AppleLink CROSE) resisted learning anything about computers in college, where she majored in math, but she couldn't escape them in the real world. First she used gigantic IBM machines at her statistical research job in Manhattan. But California beckoned, and with it came terminals spewing yellow paper and, eventually, computers with display screens. This was a big "Wow!" at first, but now Caroline gets her excitement from non-computer endeavors, such as travel. On her trip this year to the Big Island of Hawaii, she kayaked among (breaching) humpback whales and watched lava flow down cliffs and into the Pacific. Hard to top that!

LETTERS

FLOATING WINDOWS AGAIN

I'd like to use the library of functions for floating windows described in *develop* Issue 15 by Dean Yu (updated on Issue 21's CD). I'm using CodeWarrior 5.5, and when I try to compile the sample project (or any other project that includes the WindowExtensions.h file) I get a "WindowRef redeclared" error. There seems to be a conflict with the universal headers.

Before I try to get rid of this error myself (and probably make everything wrong), I thought I'd ask if you could suggest a simple and clean solution.

— Fred Klein

On this issue's CD is a new version of the floating windows library that fixes this problem, and others. The problem was that Apple finally "caught up" with Dean and defined things in the universal headers that he had defined, in his forward-looking way, back when he first wrote the article.

Also on the CD you'll find an even newer version of the library that compiles with STRICT_WINDOWS defined. This necessitated a complete rewrite of some portions of the code, so consider it risky. Please try it and send me any bugs you find!

— Dave Johnson

POWERPC ASSEMBLY NITS

Great article on PowerPC™ assembly language in *develop* Issue 21! It was clear, and I learned a lot reading it. But I have two nitpicks. On page 27 you show glue code for a cross-TOC call. The second instruction should be

```
stw    RTOC,20(SP)
```

And the third instruction has a typo in it. It should be

```
lwz    r0,0(r12)
```

— David Shayer

Thanks for catching these. The interesting thing is that the second instruction appears that (wrong) way in the PPCAsm manual. Whoops!

— Dave Evans

UP ON THE DOWNSIDE

I just wanted to tell you that I really liked the Veteran Neophyte column in Issue 21, about the downside of programming. It struck a nerve with me. The thing that goes through my mind whenever I sit down to write some code is "There has to be a better way!" Alas, by the time there is a better way, I will probably have moved on to some other profession.

— Jamie Osborne

Your Veteran Neophyte column on the pains of programming really struck a nerve (and not just because I have carpal tunnel syndrome). I often spend a while putting things on paper, only to abandon the project once I become convinced that I've figured out the solution and its implementation would just be hours and hours of typing. Sort of meta-programming.

— Tom Busey

I just finished reading the Veteran Neophyte columns in Issue 17 and Issue 21, "Why We Do It" and "The Downside." They were given to me by

KEEP US ON OUR TOES!

We welcome your nitpicking letters to the editors, especially regarding articles published in *develop*. Letters should be addressed to Caroline Rose — or, if technical *develop*-related questions, to Dave Johnson — at AppleLink CROSE or JOHNSON.DK. Or you can write to Caroline or Dave at Apple Computer, Inc., 1 Infinite Loop, M/S 303-4DP, Cupertino, CA 95014. All letters should include your name and company name as well as your address and phone number. Letters may be excerpted or edited for clarity (or to make them say what we wish they did).

a friend who is an avid programmer. The type of things you described sounded *just* like my friend; I think he showed the columns to me to explain why every time I see him he's sitting in front of the computer, and why he stays up till all hours of the morning working on programs that end up frustrating him.

I thought I should let you know that your columns were appreciated not only by those who program, but by those who are close to programmers and wonder sometimes what unseen force has gotten hold of them and sucked them into their work.

— Greta Meussling

The "Downside" column seems to have hit home with many people; I got a lot *of comments about it. It's nice to be assured that I'm not the only reluctant programmer in the world, and that I'm not the only one who thinks there ought to be a better way.*

— Dave Johnson

ACROBAT: PRETTY DARN FINE

This probably isn't the first time you've heard this, but how about offering *develop* in Acrobat (PDF format) as well? For me, Acrobat is more convenient than Apple DocViewer as an application and, most important, its files are a lot smaller. I routinely convert *develop* to PDF and then add PDF hyperlinks and bookmarks. For one issue I converted, for instance, the DocViewer version (without the index) is 2.9 meg, while the PDF version is only 770K. It's even smaller than the StuffIt version of the DocViewer document (1.2 meg). And the onscreen appearance is identical.

I still like the HTML versions for their immediacy, but for true WYSIWYG, low conversion effort, and small file size, you can't beat PDF.

— Shannon Spires

We agree with you. You'll notice that on this issue's CD, every issue of develop *has been converted to Acrobat — along with all the other files on the CD that used to be in Apple DocViewer format. Enjoy!*

— Caroline Rose

UP ON THE WEB

Thanks for making both *develop* and *Apple Directions* available on the World Wide Web. We're on a very tight budget and can't afford a subscription at this time. The online versions allow us to access the information and still come out with a product on budget.

— Mattias Fornander

I'm a student who reads *develop* online via the Internet through UCLA's (UNIX®) workstations. Your putting *develop* on the World Wide Web is great! Even though the comfort of reading (and printing) *develop* online will never equal the ease of the regular version, I don't have to fight with ftp and MS-DOS floppy disks to read your magazine. So please continue to publish *develop* in HTML.

IMHO, your magazine is a service to the Mac developer community, and you would help Apple by letting every possible programmer access it without hassle. Thanks for this effort.

— Eric Gouriou

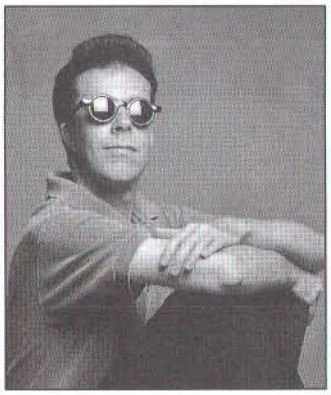
We've got articles from some issues of develop *on the World Wide Web now (at http://www.apple.com, in the Developer Services area) and are working on putting more up there. This kind of feedback helps make it happen — so thanks for writing.*

Readers of the online version: Don't confuse printed develop *with the monthly Apple Developer Mailing; a subscription to the monthly mailing (which includes a CD that has* develop *on it) is rather costly, but it costs only $30 for four quarterly printed issues of* develop *(with Bookmark CD). See the inside front cover of this issue for ordering information. (Sorry, I couldn't resist this opportunity for a plug!)*

— Caroline Rose

Music the Easy Way: The QuickTime Music Architecture

Music has become cheap and plentiful on the Macintosh, and many applications are now making "casual" use of music. With the QuickTime Music Architecture, or QTMA, including music in your application has never been simpler. Its API is straightforward and easy to use, and you don't need intimate knowledge of MIDI protocols or channel and voice numberings. Nor do you need an external MIDI device; QTMA can play music directly out of the Macintosh's built-in speakers. And QTMA is widely available — it's on every Macintosh that has QuickTime 2.0 (or later) installed.



DAVID VAN BRINK

The QuickTime Music Architecture is perfect for adding a little bit of music to your application. It has a set of well-supported high-level calls for playing musical notes and sequences, it deals with MIDI protocols so that your application doesn't have to, and it handles timing for entire tunes. With QTMA, you can specify musical instruments independent of device, and play music either directly out of built-in speakers or through a MIDI synthesizer.

QTMA first became available with QuickTime 2.0 and offers some new features in QuickTime 2.1, which should be available through APDA by the time you read this. The code in this article is written for version 2.1; minor changes will be required for 2.0. (Before making use of the QuickTime 2.1 features, your code should call Gestalt with the gestaltQuickTimeVersion selector and check the version number returned.)

This article shows how your application can use QTMA to play individual notes, sequences of notes composed on the fly, or prescored sequences, and how to read input from external MIDI devices. This issue's CD contains all the sample code and a THINK C project to build and run it. We'll start with a look at QTMA in relation to other ways of supporting music on the Macintosh; then we'll get down to business and play some music with QTMA.

QTMA IN CONTEXT — A LOOK AT MUSIC AND MIDI SUPPORT ON THE MACINTOSH

Support for MIDI and musical applications on the Macintosh platform has a somewhat checkered history. Developers have been faced with such options as writing their own serial drivers, using the MIDI Manager, or using third-party operating system extensions such as the Open Music System (OMS, formerly Opcode MIDI

DAVID VAN BRINK lives in a tiny experimental habitat overlooking the Denny's parking lot in Santa Cruz, California. He experiences life at 14,400 bits per second. See http://www.srm.com for more information.•

System) and the Free MIDI System (FMS) from Mark of the Unicorn. None of these are practical for adding just a little music to your application.

Writing a serial driver to send MIDI output to an Apple MIDI adapter or to any third-party MIDI adapter isn't that complicated if you enjoy writing low-level code to access hardware registers on the SCC serial chip. I say this in all seriousness: that kind of code really is fun to write! But it's not the best way to do things, because changes in the OS and hardware can render your work useless. And writing the low-level serial code for MIDI input has additional complexities, primarily because of the interrupt timings in many parts of the Mac OS.

The MIDI Manager is a slightly better tool to use for MIDI input and output. Unfortunately, Apple's support for this product has been less than consistent, and the MIDI Manager itself has some inherent performance limitations, though these are less critical on faster hardware (68030 processor or better).

Both OMS and FMS are quite appropriate for professional music scoring and editing products. Among the facilities that these extensions provide is a "studio configuration"; this lets the user describe to the system the various MIDI devices attached to the computer so that different applications can access them.

All of these options have drawbacks for making casual use of music: you have to access an external MIDI device, which most users don't have, and you have to use MIDI protocols to talk to that device. QTMA frees you from both of these constraints. It also frees you from needing to know a lot about MIDI itself; if you want to know anyway, check out the information in "A MIDI Primer."

QTMA'S BASIC COMPONENTS

QTMA is implemented in three easy pieces, as QuickTime components for playing individual notes, playing tunes (sequences of notes), and driving MIDI devices.

- The *note allocator component* is used to play individual notes. The calling application can specify which musical instrument sound to use and exactly which musical synthesizer to play the notes on. The note allocator component also includes a utility that allows the user to pick the instrument.
- The *tune player component* can accept entire sequences of musical notes and play them from start to finish, asynchronously, with no further need for application intervention. This is handy if you'd like to play some infernally irritating little melody, or perhaps threnody, during each game level of Boom Three Dee or whenever.
- Individual *music components* act as device drivers for each type of synthesizer attached to a particular computer. Two music components are provided with QuickTime 2.0: the software synthesizer component, to play music out of the built-in speaker, and the General MIDI component, to play music on a General MIDI device attached to a serial port. QuickTime 2.1 supports a small number of other popular synthesizers as well.

PLAYING NOTES WITH THE NOTE ALLOCATOR

Playing a few notes with the note allocator component is simple. To play notes that have a piano-like sound, you need to open up the note allocator component, allocate a *note channel* with a request for piano, and play. That's it! If you're feeling like a particularly well-behaved software engineer, you might dispose of the note channel and close the note allocator component when you're done. We'll get to the code in a moment; first we'll look at some important related data structures.

A MIDI PRIMER

MIDI, or Musical Instrument Digital Interface, uses a serial protocol and a standard 5-pin connector that you'll find on professional electronic music gear made after 1985 or so. The connector's relatively large size, about half an inch in diameter, was chosen so that it could withstand the rigors of the road — in other words, so that even drummers could plug it in.

Because MIDI cables can carry signals in only one direction, synthesizers have separate connectors for MIDI input and MIDI output. (This differs from modem cables, which carry signals in both directions.)

MIDI is a serial protocol running at 31250 baud, 8 data bits, 1 stop bit, no parity. The command structure for a MIDI stream is simple: each byte is either a *status byte* or a *data byte*.

A status byte establishes a mode for interpreting the data bytes that follow it. The high bit is set, and the next three bits indicate the type of status byte. The low four bits are typically used to specify a *MIDI channel*. Thus MIDI can address up to 16 unique channels, each of which may play a different musical instrument sound. Later extensions to MIDI let you address more channels through the use of escape codes and bank switching.

The most common status message is the Play Note message, which has a value of 0x90 plus the MIDI channel number. Each note is defined by a pitch and velocity. The pitch is an integer from 0 to 127, where 60 is musical middle C (61 is C sharp, 59 is B, 72 is the C above middle C, and so on). The velocity is an integer from 0 to 127 that describes how loud to play the note; 64 is average loudness, 127 is very loud, 1 is nearly inaudible, and 0 means to stop playing the note.

So, to play a C-major chord on MIDI channel 0, you send the seven bytes 0x90 0x3C 0x40 0x40 0x40 0x43 0x40 to begin the sound. After a suitable interval, you send 0x90 0x3C 0x00 0x40 0x00 0x43 0x00 to silence it.

All of this is exactly the sort of stuff you don't need to know if you use the QuickTime Music Architecture for your music-playing needs. But you just can't know too many useless facts, right?

NOTE-RELATED DATA STRUCTURES

A note channel is analogous to a sound channel in that you allocate it, issue commands to it to produce sound, and close it when you're done. To specify details about the note channel, you use a data structure called a NoteRequest (see Listing 1). The NoteRequestInfo structure in the NoteRequest is new in QuickTime 2.1; it simply encapsulates the first few fields of the old NoteRequest structure and splits the first of those fields into two, **flags** and **reserved** (which are decribed in the documentation accompanying the QuickTime 2.1 release).

The next two fields specify the probable *polyphony* that the note channel will be used for. Polyphony means, literally, many sounds. A polyphony of 5 means that five notes can be playing simultaneously. The polyphony field enables QTMA to make sure that the allocated note channel can play all the notes you'll need. The typicalPolyphony field is a fixed-point number that should be set to the average number of voices the note channel will play; it may be whole or fractional. Some music components use this field to adjust the mixing level for a good volume.

The ToneDescription structure is used throughout QTMA to specify a musical instrument sound in a device-independent fashion. This structure's synthesizerType and synthesizerName fields can request a particular synthesizer to play notes on. Usually, they're set to 0, meaning "choose the best General MIDI synthesizer." The gmNumber field indicates the General MIDI (GM) instrument or drum kit sound, which may be any of 135 such sounds that are supported by many synthesizer manufacturers. (All these sounds are available on a General MIDI Sound Module.) The GM instruments are numbered 1 through 128, and the seven drum kits are numbered 16385 and higher. A complete list of instrument and drum kit numbers is provided in Table 1. For synthesizers that accept sounds outside the GM library, you

Listing 1. Note-related data structures

```
struct NoteRequest {
   NoteRequestInfo   info;    // · in post-QuickTime 2.0 only
   ToneDescription   tone;
};

struct NoteRequestInfo {
   UInt8    flags;
   UInt8    reserved;
   short    polyphony;
   Fixed    typicalPolyphony;
};

struct ToneDescription {
   OSType   synthesizerType;
   Str31    synthesizerName;
   Str31    instrumentName;
   long     instrumentNumber;
   long     gmNumber;
};
```

can use the instrumentName and instrumentNumber fields to specify some other sound.

THE NOTE-PLAYING CODE

The routine in Listing 2 plays notes in a piano-like sound with the note allocator component. We start by calling OpenDefaultComponent to open up the component. If this routine returns 0, the component wasn't opened, most likely because QTMA wasn't present.

Next we fill in the NoteRequestInfo and ToneDescription structures, calling the note allocator's NAStuffToneDescription routine and passing it the GM instrument number for piano. This routine fills in the gmNumber field and also fills in the other ToneDescription fields with sensible values, such as the instrument's name in text form in the instrumentName field. (The routine can be useful for converting a GM instrument number to its text equivalent.)

After allocating the note channel with NANewNoteChannel, we call NAPlayNote to play each note. Notice the last two parameters to NAPlayNote:

```
ComponentResult NAPlayNote(NoteAllocator na, NoteChannel nc,
   long pitch, long velocity);
```

The value of the pitch parameter is an integer from 1 to 127, where 60 is middle C, 61 is C sharp, and 59 is C flat, or B. Similarly, 69 is concert A, and is played at a nominal audio frequency of 440 Hz. The velocity parameter's value is also an integer from 1 to 127, or 0. A velocity of 1 corresponds to just barely touching the musical keyboard, and 127 indicates that the key was struck as hard as possible. Different velocities produce tones of different volumes from the synthesizer. A velocity of 0 means the key was released; the note stops or fades out, as appropriate to the kind of sound being played. Here we stop the notes after delaying an appropriate amount of time with a call to the Delay routine.

Table 1. The General MIDI instruments and drum kits

Piano

- 1 Acoustic Grand Piano
- 2 Bright Acoustic Piano
- 3 Electric Grand Piano
- 4 Honky-tonk Piano
- 5 Rhodes Piano
- 6 Chorused Piano
- 7 Harpsichord
- 8 Clavinet

Chromatic Percussion

- 9 Celesta
- 10 Glockenspiel
- 11 Music Box
- 12 Vibraphone
- 13 Marimba
- 14 Xylophone
- 15 Tubular bells
- 16 Dulcimer

Organ

- 17 Hammond Organ
- 18 Percussive Organ
- 19 Rock Organ
- 20 Church Organ
- 21 Reed Organ
- 22 Accordion
- 23 Harmonica
- 24 Tango Accordion

Guitar

- 25 Acoustic Nylon Guitar
- 26 Acoustic Steel Guitar
- 27 Electric Jazz Guitar
- 28 Electric Clean Guitar
- 29 Electric Muted Guitar
- 30 Overdriven Guitar
- 31 Distortion Guitar
- 32 Guitar Harmonics

Bass

- 33 Acoustic Fretless Bass
- 34 Electric Bass Fingered
- 35 Electric Bass Picked
- 36 Fretless Bass
- 37 Slap Bass 1
- 38 Slap Bass 2
- 39 Synth Bass 1
- 40 Synth Bass 2

Strings and Orchestra

- 41 Violin
- 42 Viola
- 43 Cello
- 44 Contrabass
- 45 Tremolo Strings
- 46 Pizzicato Strings
- 47 Orchestral Harp
- 48 Timpani

Ensemble

- 49 Acoustic String Ensemble 1
- 50 Acoustic String Ensemble 2
- 51 SynthStrings 1
- 52 SynthStrings 2
- 53 Aah Choir
- 54 Ooh Choir
- 55 Synth Vox
- 56 Orchestra Hit

Brass

- 57 Trumpet
- 58 Trombone
- 59 Tuba
- 60 Muted Trumpet
- 61 French Horn
- 62 Brass Section
- 63 Synth Brass 1
- 64 Synth Brass 2

Reed

- 65 Soprano Sax
- 66 Alto Sax
- 67 Tenor Sax
- 68 Baritone Sax
- 69 Oboe
- 70 English Horn
- 71 Bassoon
- 72 Clarinet

Pipe

- 73 Piccolo
- 74 Flute
- 75 Recorder
- 76 Pan Flute
- 77 Bottle Blow
- 78 Shakuhachi
- 79 Whistle
- 80 Ocarina

Synth Lead

- 81 Square Wave
- 82 Saw Wave
- 83 Calliope
- 84 Chiffer
- 85 Charang
- 86 Solo Vox
- 87 5th Saw Wave
- 88 Bass and Lead

Synth Pad

- 89 Fantasy
- 90 Warm
- 91 Polysynth
- 92 Choir
- 93 Bowed
- 94 Metal
- 95 Halo
- 96 Sweep

Synth Effect

- 97 Ice Rain
- 98 Sound Tracks
- 99 Crystal
- 100 Atmosphere
- 101 Brightness
- 102 Goblins
- 103 Echoes
- 104 Space

Ethnic

- 105 Sitar
- 106 Banjo
- 107 Shamisen
- 108 Koto
- 109 Kalimba
- 110 Bagpipe
- 111 Fiddle
- 112 Shanai

Percussive

- 113 Tinkle Bell
- 114 Agogo
- 115 Steel Drums
- 116 Woodblock
- 117 Taiko Drum
- 118 Melodic Tom
- 119 Synth Drum
- 120 Reverse Cymbal

Sound Effects

- 121 Guitar Fret Noise
- 122 Breath Noise
- 123 Seashore
- 124 Bird Tweet
- 125 Telephone Ring
- 126 Helicopter
- 127 Applause
- 128 Gunshot

GM Drum Kits

- 16385 Standard Kit
- 16393 Room Kit (a memory-reduced version of the Standard Kit)
- 16401 Power Kit
- 16409 Electronic Kit
- 16410 Analog Kit
- 16425 Brush Kit
- 16433 Orchestra Kit

- Bullets indicate the instruments and drum kits that are available for playing on the built-in synthesizer.

Listing 2. Playing notes with the note allocator component

```
void PlaySomeNotes(void)
{
   NoteAllocator     na;
   NoteChannel       nc;
   NoteRequest       nr;
   ComponentResult   thisError;
   long              t, i;

   na = 0;
   nc = 0;

   // · Open up the note allocator.
   na = OpenDefaultComponent(kNoteAllocatorType, 0);
   if (!na)
      goto goHome;

   // · Fill out a NoteRequest using NAStuffToneDescription to help, and
   // · allocate a NoteChannel.
   nr.info.flags = 0;
   nr.info.reserved = 0;
   nr.info.polyphony = 2;     // · simultaneous tones
   nr.info.typicalPolyphony = 0x00010000;   // · usually just one note
   thisError = NAStuffToneDescription(na, 1, &nr.tone);  // · 1 is piano
   thisError = NANewNoteChannel(na, &nr, &nc);
   if (thisError || !nc)
      goto goHome;

   // · If we've gotten this far, OK to play some musical notes. Lovely.
   NAPlayNote(na, nc, 60, 80);  // · middle C at velocity 80
   Delay(40, &t);               // · delay 2/3 of a second
   NAPlayNote(na, nc, 60, 0);   // · middle C at velocity 0: end note
   Delay(40, &t);               // · delay 2/3 of a second

   // · Obligatory do-loop of rising tones
   for (i = 60; i <= 84; i++) {
      NAPlayNote(na, nc, i, 80);   // · pitch i at velocity 80
      NAPlayNote(na, nc, i+7, 80); // · pitch i+7 (musical fifth) at
                                   // · velocity 80
      Delay(10, &t);               // · delay 1/6 of a second
      NAPlayNote(na, nc, i, 0);    // · pitch i at velocity 0: end note
      NAPlayNote(na, nc, i+7, 0);  // · pitch i+7 at velocity 0:
                                   // · end note
   }

goHome:
   if (nc)
      NADisposeNoteChannel(na, nc);
   if (na)
      CloseComponent(na);
}
```

ROGER SHEPARD'S MELODY

In Listing 2, if you replace the code in the section labeled "Obligatory do-loop of rising tones" with the following code, you'll receive a secret treat.

```
i = 0;
while (!Button()) {
   long  j, v;
   for (j = i % 13; j < 128; j+=13) {
      v = j < 64 ? j * 2 : (127 - j) * 2;
      NAPlayNote(na, nc, j, v);
   }
   Delay(13, &t);
   for (j = i % 13; j < 128; j+=13)
      NAPlayNote(na, nc, j, 0);
   i++;
}
```

This snappy little melody was discovered by psychologist Roger Shepard in the 1960s.

Finally, being well behaved, we dispose of the note channel and close the note allocator component.

LETTING THE USER PICK THE INSTRUMENT

Rather than specify the instrument sound itself, your application may want to let the user pick it. For this purpose, a nifty instrument picker utility is provided in the note allocator component. The instrument picker dialog, shown in Figure 1, enables users to choose musical instruments from the available synthesizers and sounds.

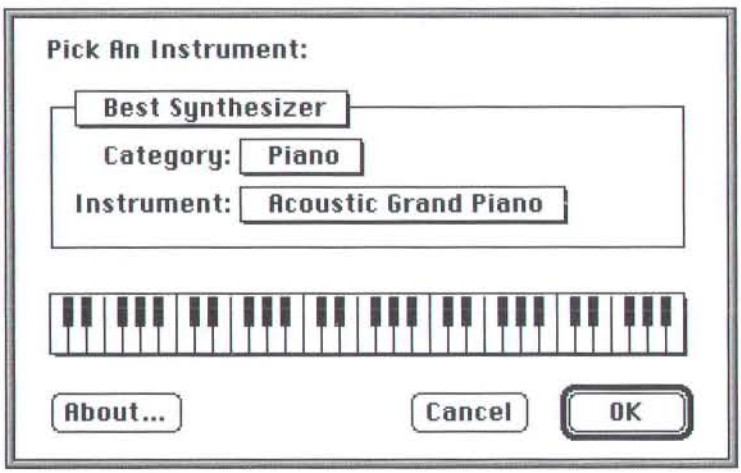


Figure 1. The instrument picker dialog

The routine in Listing 3 shows one way that your application can use the instrument picker. It's nearly identical to the code in Listing 2, except that the NAPickInstrument routine is called right after the call to NAStuffToneDescription. As in Listing 1, NAStuffToneDescription fills out a ToneDescription record for a particular GM instrument number; NAPickInstrument then invokes the instrument

picker dialog and alters the passed ToneDescription to whatever instrument the user selects.

Listing 3. Using the instrument picker

```
void PickThenPlaySomeNotes(void)
{
   ...   // · declarations and initialization

   // · Open up the note allocator.
   ...

   // · Fill out a NoteRequest using NAStuffToneDescription to help,
   // · call NAPickInstrument, and allocate a NoteChannel.
   nr.info.flags = 0;
   nr.info.reserved = 0;
   nr.info.polyphony = 2;     // · simultaneous tones
   nr.info.typicalPolyphony = 0x00010000;
   thisError = NAStuffToneDescription(na, 1, &nr.tone);  // · 1 is piano
   thisError = NAPickInstrument(na, nil, "\pPick An Instrument:",
                  &nr.tone, 0, 0, nil, nil);
   if (thisError)
      goto goHome;
   thisError = NANewNoteChannel(na, &nr, &nc);
   if (thisError || !nc)
      goto goHome;

   // · Play some musical notes.
   ...

   // · Obligatory do-loop of rising tones
   ...

goHome:
   ...      // · Dispose of the NoteChannel and close the component.
}
```

ADDING EXPRESSIVENESS WITH CONTROLLERS

There's much more to music than simply playing the right notes at the right times. Although your code can simulate only a scant fraction of the expressiveness of a skillfully played acoustic instrument, there are certain things the note allocator component lets you do that help make your computer-synthesized music sound more interesting.

As we've already seen, the NAPlayNote routine has parameters for specifying pitch and velocity, the latter determining the volume of the note; changes in these parameter values can affect the expressiveness of your music. You can also add expressiveness to whatever notes are being played by using QTMA's *controllers*. A controller is a parameter that's set independently of the notes being played, with a call to the NASetController routine:

```
ComponentResult NASetController(NoteAllocator na, NoteChannel nc,
   long controllerNumber, long controllerValue);
```

Two simple controllers are the *pitch bend controller* and the *volume controller*. The pitch bend controller alters the frequency of any notes being played. It's like the whammy-bar on an electric guitar, which tightens or loosens all the strings simultaneously. The volume controller affects the sound of all notes similarly to the way key velocity affects the sound of individual notes.

Let's look at some source code that uses the pitch bend controller (Listing 4). This routine plays a major-fifth interval for a half second, "bends" it up by three semitones, holds it a half second, and then bends it back down to its original pitch.

Listing 4. Using the pitch bend controller

```
void PlaySomeBentNotes(void)
{
   ...   // · declarations and initialization

   // · Open up the note allocator.
   ...

   // · Fill out a NoteRequest using NAStuffToneDescription to help, and
   // · allocate a NoteChannel.
   ...

   // · If we've gotten this far, OK to play some musical notes. Lovely.
   NAPlayNote(na, nc, 60, 80);  // · middle C at velocity 80
   NAPlayNote(na, nc, 67, 60);  // · G at velocity 60
   Delay(30, &t);

   // · Loop through differing pitch bendings.
   for (i = 0; i <= 0x0300; i+=10) {     // · bend 3 semitones
      NASetController(na, nc, kControllerPitchBend, i);
      Delay(1, &t);
   }
   Delay(30, &t);
   for (i = 0x0300; i >= 0; i-=10) {     // · bend back to normal
      NASetController(na, nc, kControllerPitchBend, i);
      Delay(1, &t);
   }
   Delay(30, &t);
   NAPlayNote(na, nc, 60, 0);    // · middle C off
   NAPlayNote(na, nc, 67, 0);    // · G off

goHome:
   ...      // · Dispose of the NoteChannel and close the component.
}
```

Most QuickTime controller values are 16-bit signed fixed-point numbers (where the lower eight bits are fractional) and have a range of 0 to 127, with a default value of 0. However, the pitch bend controller has a range of -127 to 127, and the volume controller has a default value of 127, or maximum volume.

The *pan controller* has a slightly different definition from the other controllers. "Pan" refers to the position of the sound in the stereo field. Most synthesizers have audio

output for left and right; on such synthesizers, the pan value is interpreted as follows: The default pan position (usually centered) is specified by a value of 0 to the pan controller. To position the sound arbitrarily, values between 1 (0x0100) and 2 (0x0200) are used to range between left and right, respectively. For synthesizers with *n* outputs, values between 1 and *n* are used to pan between each adjacent pair of outputs. Note that the built-in synthesizer doesn't currently support panning.

BUILDING A TUNE

As mentioned earlier, an application can use the tune player component to play entire sequences of notes, or tunes. Applications often find it useful to play a tune that has been precomposed and stored in the application; other times, it may be useful to construct a tune at run time and then play it. In either case, the application must first build the tune. Here we'll take a look at the format of a tune and the routines and macros we use for building one.

THE FORMAT OF A TUNE

The format for tunes is a series of long words, subdivided into bitfields. Your application needs to build a *tune header* and *tune sequence* made up of different types of "events." The tune header contains one or more *note request events*, each a NoteRequest data structure with some encapsulating long words. The tune sequence is made up of *note events* that specify notes and durations, controller changes, and so on, as well as *rest events*; it's the musical score.

In the tune header, each note request event has the structure shown in Figure 2. (It's actually a *general event*, of the note request subtype.) Thus the first word is 0x*Fnnn*0017, where *nnn* is the part number, and the last word is 0xC0010017. The part number is referred to later on by note events in the tune sequence. For example, given a header than contains a note request event specifying part 3, subsequent note events that specify part 3 will play in a note channel allocated according to that NoteRequest.

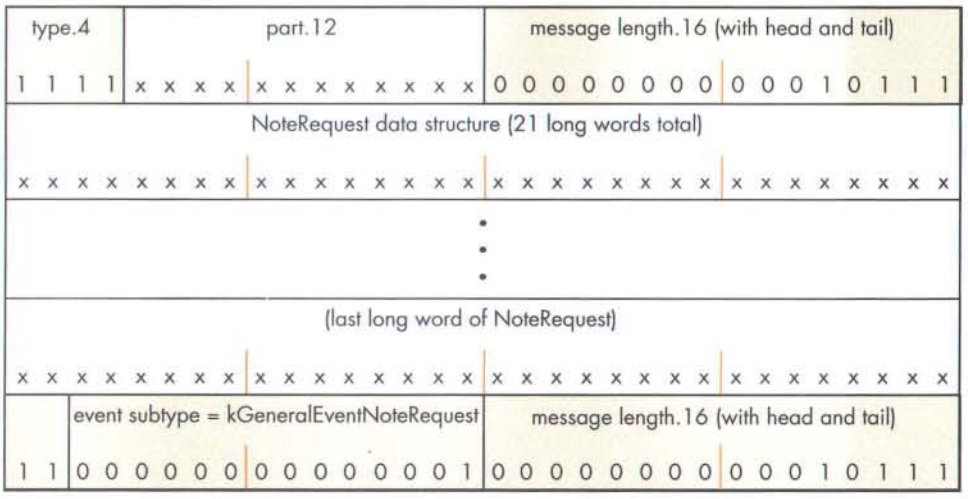
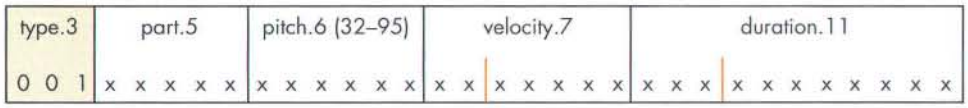


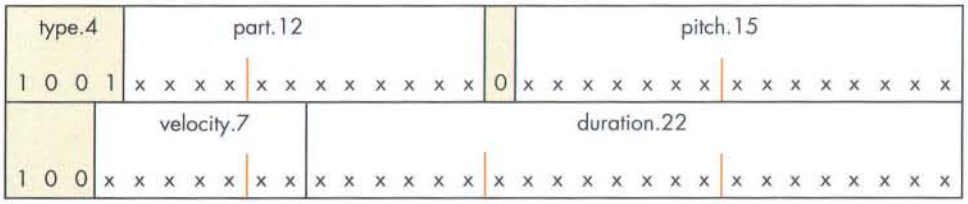
Figure 2. A note request event

In the tune sequence, each note event includes the part, pitch, velocity, and duration of the note; a rest event specifies only a duration (see Figure 3). A note event can have either a short or an extended format. In a short note event, the pitch is limited to the range 32 to 95 (which covers most musical notes) and the part number must be less

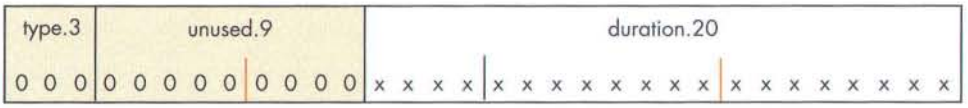
than 32. If either of these ranges is too small, or if you want to use a fixed-point pitch value or a very long duration, the extended note format may be used. Much of the time you can use the short format, to save space.



A short note event



An extended note event



A rest event

Figure 3. Note and rest events

Both headers and sequences end with a *marker event* containing all zeroes (equivalent to 0x60000000), shown in Figure 4.

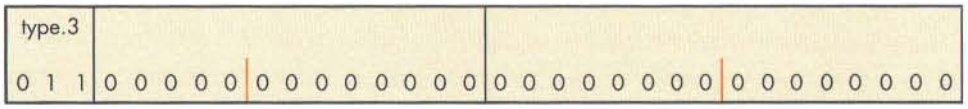


Figure 4. A marker event

THE TUNE-BUILDING CODE

Our sample code includes routines for building the tune header and tune sequence. These routines use some handy event-stuffing macros that are defined in the file QuickTimeComponents.h, and all have the form _StuffSomething(arguments). BuildTuneHeader (Listing 5) uses the following macro:

```
_StuffGeneralEvent(w1, w2, part, subtype, length);
```

The _StuffGeneralEvent macro fills in the head and tail long words of a particular type of general event — in our case, a note request event. Its arguments are, in order: the head and tail long words; the part number; the event subtype (kGeneralEventNoteRequest for a note request event); and the length in long words of the entire event, counting the head and tail. Note that the first two arguments are the head and tail themselves, not pointers — the macro expands to a direct assignment of these arguments.

BuildTuneSequence (Listing 6) uses the _StuffNoteEvent and _StuffRestEvent macros.

Listing 5. BuildTuneHeader

```
#define kNoteRequestHeaderEventLength \
            (sizeof(NoteRequest) /sizeof(long) + 2)    // · long words
#define our_header_length \
   ((2 * kNoteRequestHeaderEventLength + 1)) * sizeof(long)    // · bytes
unsigned long *BuildTuneHeader(void)
{
   unsigned long     *header, *w, *w2;
   NoteRequest       *nr;
   NoteAllocator     na;   // · just for the NAStuffToneDescription call
   ComponentResult   thisError;

   header = 0;
   na = 0;

   // · Open up the note allocator.
   na = OpenDefaultComponent(kNoteAllocatorType, 0);
   if (!na)
      goto goHome;

   // · Allocate space for the tune header, rather inflexibly.
   header = (unsigned long *) NewPtrClear(our_header_length);
   if (!header)
      goto goHome;
   w = header;

   // · Stuff request for piano polyphony 4.
   w2 = w + kNoteRequestHeaderEventLength - 1; // · last long word of
                                               // · note request event
   _StuffGeneralEvent(*w, *w2, 1, kGeneralEventNoteRequest,
                      kNoteRequestHeaderEventLength);
   nr = (NoteRequest *)(w + 1);
   nr->info.flags = 0;
   nr->info.reserved = 0;
   nr->info.polyphony = 4;    // · simultaneous tones
   nr->info.typicalPolyphony = 0x00010000;
   thisError = NAStuffToneDescription(na, 1, &nr->tone); // · 1 is piano
   w += kNoteRequestHeaderEventLength;

   // · Stuff request for violin polyphony 3.
   w2 = w + kNoteRequestHeaderEventLength - 1; // · last long word of
                                               // · note request event
   _StuffGeneralEvent(*w, *w2, 2, kGeneralEventNoteRequest,
                      kNoteRequestHeaderEventLength);
   nr = (NoteRequest *)(w + 1);
   nr->info.flags = 0;
   nr->info.reserved = 0;
   nr->info.polyphony = 3;    // · simultaneous tones
   nr->info.typicalPolyphony = 0x00010000;
   thisError = NAStuffToneDescription(na, 41, &nr->tone);  // · violin
   w += kNoteRequestHeaderEventLength;
   *w++ = 0x60000000;      // · end-of-sequence marker
```

(continued on next page)

Listing 5. BuildTuneHeader *(continued)*

```
goHome:
   if (na)
      CloseComponent(na);
   return header;
}
```

```
_StuffNoteEvent(w, part, pitch, volume, duration);
```

The _StuffNoteEvent macro fills in a note event. Its arguments are, in order: the long word to stuff; the part number; the pitch (where, as usual, 60 is middle C); the volume (velocity); and the duration (usually specified in 600ths of a second). The pitch must be between 32 and 95, and the part number must be less than 32. For values outside these ranges, a fixed-point pitch value, or a very long duration, use _StuffXNoteEvent.

```
_StuffXNoteEvent(w1, w2, part, pitch, volume, duration);
```

The _StuffXNoteEvent macro is for extended note events. It's identical to _StuffNoteEvent except that it provides larger ranges for pitch, part, and duration, and the event itself takes two long words.

```
_StuffRestEvent(w, restDuration);
```

The _StuffRestEvent macro fills in a rest event. It takes two arguments: the long word to stuff and the duration of the rest.

Listing 6. BuildTuneSequence

```
#define kNoteDuration 240     // · in 600ths of a second
#define kRestDuration 300     // · in 600ths -- tempo will be 120 bpm

#define our_sequence_length (22 * sizeof(long))      // · bytes
#define our_sequence_duration (9 * kRestDuration)    // · 600ths

unsigned long *BuildTuneSequence(void)
{
   unsigned long  *sequence, *w;

   // · Allocate space for the tune sequence, rather inflexibly.
   sequence = (unsigned long *) NewPtrClear(our_sequence_length);
   if (!sequence)
      goto goHome;
   w = sequence;
   _StuffNoteEvent(*w++, 1, 60, 100, kNoteDuration);    // · piano C
   _StuffRestEvent(*w++, kRestDuration);
   _StuffNoteEvent(*w++, 2, 60, 100, kNoteDuration);    // · violin C
   _StuffRestEvent(*w++, kRestDuration);
   _StuffNoteEvent(*w++, 1, 63, 100, kNoteDuration);    // · piano
   _StuffRestEvent(*w++, kRestDuration);
```

(continued on next page)

Listing 6. BuildTuneSequence *(continued)*

```
    _StuffNoteEvent(*w++, 2, 64, 100, kNoteDuration);    // · violin
    _StuffRestEvent(*w++, kRestDuration);

    // · Make the 5th and 6th notes much softer, just for fun.
    _StuffNoteEvent(*w++, 1, 67, 60, kNoteDuration);     // · piano
    _StuffRestEvent(*w++, kRestDuration);
    _StuffNoteEvent(*w++, 2, 66, 60, kNoteDuration);     // · violin
    _StuffRestEvent(*w++, kRestDuration);
    _StuffNoteEvent(*w++, 1, 72, 100, kNoteDuration);    // · piano
    _StuffRestEvent(*w++, kRestDuration);
    _StuffNoteEvent(*w++, 2, 73, 100, kNoteDuration);    // · violin
    _StuffRestEvent(*w++, kRestDuration);
    _StuffNoteEvent(*w++, 1, 60, 100, kNoteDuration);    // · piano
    _StuffNoteEvent(*w++, 1, 67, 100, kNoteDuration);    // · piano
    _StuffNoteEvent(*w++, 2, 63, 100, kNoteDuration);    // · violin
    _StuffNoteEvent(*w++, 2, 72, 100, kNoteDuration);    // · violin
    _StuffRestEvent(*w++, kRestDuration);
    *w++ = 0x60000000;   // · end-of-sequence marker

goHome:
    return sequence;
}
```

It's important to understand that the duration of a sequence equals the total durations of all the *rest* events. The durations within the note events don't contribute to the duration of the sequence! If two note events occur in a row, each with a duration of say 100, they'll both start at the same time, not 100 time units apart. If the next event is an end-of-sequence marker, the notes will immediately be stopped, having played for zero time units. If, however, a rest event is placed between the note events and the end marker, both notes will sound for the duration of the rest event, up to 100 time units.

PLAYING A TUNE WITH THE TUNE PLAYER

Playing a tune with the tune player component is ideal if for some reason your application will be constructing a tune at run time and then playing it. For prescored music, however, the best solution is to create a QuickTime movie containing only a music track and play it as a regular movie with the Movie Toolbox, as described below.

Using the tune player to play a tune without application intervention is straightforward, as illustrated in Listing 7. After building the tune with BuildTuneHeader and BuildTuneSequence, this routine opens up a connection to the tune player component, calls TuneSetHeader with a pointer to the header information, and then calls TuneQueue with a pointer to the sequence data. All the details of playback are taken care of by the tune player. The tune will stop playing when it reaches the end or when the tune player component is closed.

PLAYING PRESCORED MUSIC IN A QUICKTIME MOVIE

The best way to play prescored music is to create a QuickTime movie with just a music track and play it with the Movie Toolbox, which takes care of details like spooling multiple segments of sequence data from disk. This is currently the only way

Listing 7. Playing a tune with the tune player component

```
void BuildSequenceAndPlay(void)
{
   unsigned long     *header, *sequence;
   TunePlayer        tp;
   TuneStatus        ts;
   ComponentResult   thisError;

   tp = 0;
   header = BuildTuneHeader();
   sequence = BuildTuneSequence();
   if (!header || !sequence)
      goto goHome;
   tp = OpenDefaultComponent(kTunePlayerType, 0);
   if (!tp)
      goto goHome;
   thisError = TuneSetHeader(tp, header);
   thisError = TuneQueue(tp, sequence, 0x00010000, 0, 0x7FFFFFFF,
                         0, 0, 0);

   // · Wait until the sequence finishes playing or the user clicks
   // · the mouse.
spin:
   thisError = TuneGetStatus(tp, &ts);
   if (ts.queueTime && !Button())
      goto spin;     // · I use gotos primarily to shock the children.

goHome:
   if (tp)
      CloseComponent(tp);
   if (header)
      DisposePtr((Ptr) header);
   if (sequence)
      DisposePtr((Ptr) sequence);
}
```

to create QuickTime music that will also play under QuickTime for Windows. There are many tools for authoring music into Standard MIDI Files, which are then easily imported as QuickTime movies — but first let's look at the more hard-core method of creating your own sequence and header data and saving it as a QuickTime movie.

CREATING A QUICKTIME MUSIC TRACK

Creating a QuickTime music track is exactly the same as creating any other kind of track. You create or open the movie you're adding the track to, and then add a new track and a new media followed by a sample description and the sample data. For a music track, the sample description is the tune header information, and the data is one or more tune sequences. The routine in Listing 8 constructs a QuickTime movie with a music track and saves it to disk.

IMPORTING A STANDARD MIDI FILE AS A MOVIE

Most music content exists in a format called Standard MIDI File (SMF). All sequencing and composition programs have an option to Save As or Export files to

Listing 8. Creating a QuickTime music track

```
void BuildMusicMovie(void)
{
   ComponentResult      result;
   StandardFileReply    reply;
   short                resRefNum;
   Movie                mo;
   Track                tr;
   Media                me;
   unsigned long        *tune, *header;
   MusicDescription     **mdH, *md;

   StandardPutFile("\pMusic movie file name:", "\pMovie File", &reply);
   if (!reply.sfGood)
      goto goHome;
   EnterMovies();

   // · Create the movie, track, and media.
   result = CreateMovieFile(&reply.sfFile, 'TVOD', smCurrentScript,
      createMovieFileDeleteCurFile, &resRefNum, &mo);
   if (result)
      goto goHome;
   tr = NewMovieTrack(mo, 0, 0, 256);
   me = NewTrackMedia(tr, MusicMediaType, 600, nil, 0);

   // · Create a music sample description.
   header = BuildTuneHeader();
   mdH = (MusicDescription **)
      NewHandleClear(sizeof(MusicDescription) - 4 + our_header_length);
   if (!mdH)
      goto goHome;
   md = *mdH;
   md->descSize = GetHandleSize((Handle) mdH);
   md->dataFormat = kMusicComponentType;
   BlockMove(header, md->headerData, our_header_length);
   DisposePtr((Ptr) header);

   // · Get a tune, add it to the media, and then finish up.
   tune = BuildTuneSequence();
   result = BeginMediaEdits(me);
   result = AddMediaSample(me, (Handle) &tune, 0, our_sequence_length,
      our_sequence_duration, (SampleDescriptionHandle) mdH, 1, 0, nil);
   result = EndMediaEdits(me);
   result = InsertMediaIntoTrack(tr, 0, 0, our_sequence_duration,
      (1L<<16));
   result = OpenMovieFile(&reply.sfFile, &resRefNum, fsRdWrPerm);
   result = AddMovieResource(mo, resRefNum, 0, 0);
   result = CloseMovieFile(resRefNum);
   DisposePtr((Ptr) tune);
   DisposeMovie(mo);

goHome:
   ExitMovies();
}
```

this format. QuickTime has facilities for reading an SMF file and easily converting it into a QuickTime movie. (QuickTime 2.1 corrects some critical bugs in the 2.0 converter.) During any kind of conversion, the SMF file is assumed to be scored for a General MIDI device, and MIDI channel 10 is assumed to be a drum track.

The conversion to a QuickTime movie can happen in several ways. Because the conversion is implemented in a QuickTime 'eat ' component, it very often will happen automatically. Any application that uses the StandardGetFile routine to open a movie can also open 'Midi' files transparently, and can transparently paste Clipboard contents of type 'Midi' into a movie that's shown with the standard movie controller. To explicitly convert a file or handle into a movie, an application can use the Movie Toolbox routines ConvertFileToMovieFile and PasteHandleIntoMovie, respectively.

For those of you who are hard-core MIDI heads, the following two MIDI system-exclusive messages, new in QuickTime 2.1, may be useful for more precise control of the MIDI import process. (Note that QuickTime data is divided into *media samples*. Within video tracks, each video frame is considered one sample; in music tracks, each sample may contain several seconds worth of musical information.)

- F0 11 00 01 *xx yy zz* F7 sets the maximum size of each media sample to the 21-bit number *xxyyzz*. (MIDI data bytes have the high bit clear, so they have only seven bits of number.) This message can occur anywhere in an SMF file.
- F0 11 00 02 F7 marks an immediate sample break; it ends the current sample and starts a new one. All messages after a sample break message will be placed in a new media sample.

Applications can define their own system-exclusive messages of the form F0 11 7F *ww xx yy zz ... application-defined data ...* F7, where *ww xx yy zz* is the application's unique signature with the high bits cleared. This is guaranteed not to interfere with Apple's or any other manufacturer's use of system-exclusive codes.

READING INPUT FROM A MIDI DEVICE

If the user has a MIDI keyboard attached to the computer, your application can use it as an input device by calling QTMA routines that capture each event as the user triggers it.

The *default MIDI input* is whichever MIDI port the user has chosen for a General MIDI device from the QuickTime Music control panel, shown in Figure 5. (The default MIDI input can also be specified with the NASetDefaultMIDIInput call in the note allocator, but this call should be made only by music-configuration software, such as the control panel.)

An application can receive MIDI events from the default MIDI input by installing a readHook routine. This routine is called at interrupt level whenever MIDI data arrives. It's installed with the NAUseDefaultMIDIInput call (and later deinstalled with NALoseDefaultMIDIInput).

```
pascal ComponentResult NAUseDefaultMIDIInput(NoteAllocator na,
   MusicMIDIReadHookUPP readHook, long refCon, unsigned long flags);
```

The readHook routine is defined as follows:

```
typedef pascal ComponentResult (*MusicMIDIReadHookProcPtr)
   (MusicMIDIPacket *mp, long myRefCon);
```

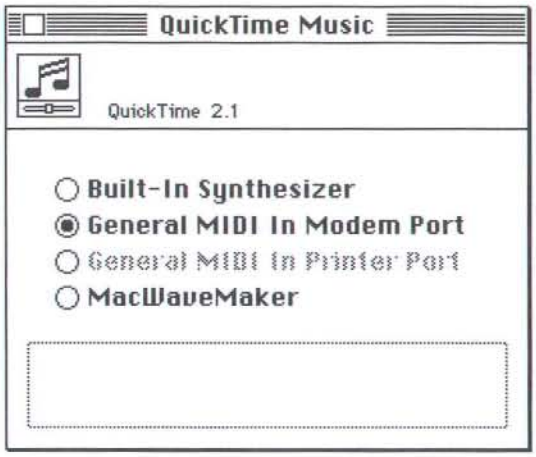


Figure 5. The QuickTime Music control panel

When the readHook routine is called, it's passed its refCon (installed with the routine) and a pointer to the MIDI packet. The MIDI packet structure is simply a list of bytes of a MIDI message, preceded by a length:

```
struct MusicMIDIPacket {
   unsigned short length;
   unsigned long  reserved;
   UInt8          data[249];
};
```

The length field is the number of bytes in the MIDI message. (If you're familiar with the MIDI Manager definition of a MIDI packet or with OMS's packet, note that their length field is different from this one: Theirs is the length of both the header and the packet data, so the minimum length would be 6; but in QuickTime's packets, the length field is only the number of bytes of MIDI data actually in the data array.)

In QuickTime 2.0, the reserved field must be set to 0, but in QuickTime 2.1, this field takes on some additional meanings (as reserved fields occasionally do). When an application is using the default MIDI input, it may occasionally lose the use of that input, such as when another application tries to use it, or if the instrument picker dialog box comes to the front. If the use of the input is lost, the reserved field will have the value kMusicPacketPortLost = 1, and the length field will be 0: no MIDI data. When the port is once again available, the readHook routine will receive a packet with the reserved field set to kMusicPacketPortFound = 2, also with no data.

The data array in the MIDI packet contains a raw MIDI message that your readHook routine will have to parse. Our example code parses only the MIDI messages for *note-on events* and *note-off events*; other messages, such as pitch-bend controls, are simply ignored.

The note-on event message has three bytes, 9*c pp vv* (in hexadecimal), where *c* is the MIDI channel that the musical keyboard is transmitting on, *pp* is a MIDI pitch from 0 to 127 (60 is middle C), and *vv* is the velocity with which the key was struck, from 1 to 127. If the velocity is 0, the message signifies a note-off event. Some devices, however, use a separate message type for note-off events; it has the form 8*c pp vv*, where *c* and *pp* are the channel and pitch, and *vv* is the velocity with which the key was released. Nobody in the world pays attention to the release velocity, so in our

example we won't either. When an *8c* message is received, we'll just set the velocity to 0 and pretend it was a *9c* message.

Listing 9 shows a readHook routine and the routine that installs it. The main routine, UseMIDIInput, allocates a note channel and then calls NAUseDefaultMIDIInput, specifying a readHook routine that parses note-on or note-off event messages. These messages are expanded into a chord that's played on the note channel. Any packet that isn't of that type — that is, doesn't contain three bytes or start with 0x8n or 0x9n — is ignored.

Listing 9. Parsing MIDI messages in the readHook routine

```
pascal ComponentResult AReadHook(MusicMIDIPacket *mp, long refCon)
{
   MIDIInputExample  *mie;
   Boolean           major;
   short             status, pitch, vel;

   mie = (MIDIInputExample *)refCon;
   if (mp->reserved == kMusicPacketPortLost)   // · port gone? make
                                               // · channel quiet
      NASetNoteChannelVolume(mie->na, mie->nc, 0);
   else if (mp->reserved == kMusicPacketPortFound)   // · port back?
                                                     // · raise volume
      NASetNoteChannelVolume(mie->na, mie->nc, 0x00010000);
   else if (mp->length == 3) {
      status = mp->data[0] & 0xF0;
      pitch = mp->data[1];
      vel = mp->data[2];
      switch (status) {
         case 0x80:
            vel = 0;
         // · Falls into case 0x90.
         case 0x90:
            major = pitch % 5 == 0;
            NAPlayNote(mie->na, mie->nc, pitch, vel);
            NAPlayNote(mie->na, mie->nc, pitch+3+major, vel);
            NAPlayNote(mie->na, mie->nc, pitch+7, vel);
            break;
      }
   }
   return noErr;
}

void UseMIDIInput(void)
{
   ComponentResult   result;
   MIDIInputExample  mie;
   NoteRequest       nr;

   mie.na = OpenDefaultComponent(kNoteAllocatorType, 0);
   if (!mie.na)
      goto goHome;
```

(continued on next page)

Listing 9. Parsing MIDI messages in the readHook routine *(continued)*

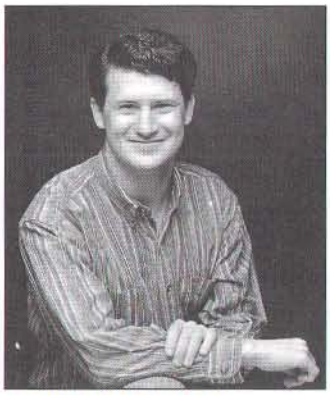
```
    nr.polyphony = 2;
    nr.typicalPolyphony = 0x00010000;
    result = NAStuffToneDescription(mie.na, 1, &nr.tone);   // · piano
    result = NANewNoteChannel(mie.na, &nr, &mie.nc);
    result = NAUseDefaultMIDIInput(mie.na, AReadHookUPP, (long) &mie, 0);
    while (!Button());
    result = NALoseDefaultMIDIInput(mie.na);

goHome:
    if (mie.na)
        CloseComponent(mie.na);    // · disposes of NoteChannel, too
}
```

GIVE QTMA A TRY

Sometimes a little music can make your application easier and more fun to use. Adding music doesn't have to be a complex task; QTMA takes care of all the hard parts, like using MIDI protocols, so you can concentrate more on the music itself. So go ahead, play some tunes and enjoy the music!

Thanks to our technical reviewers Peter Hoddie, Duncan Kennedy, Jim Nitchals, Jim Reekes, and Kent Sandvik.•



DAVID HAYWARD

PRINT HINTS

Syncing Up With ColorSync 2.0

In March of this year, Apple announced a major upgrade to the ColorSync extension and API: ColorSync 2.0. Like version 1.0, ColorSync 2.0 is a powerful color management system that allows applications and device drivers to produce consistent color across different devices. However, ColorSync 2.0 dramatically improves the quality, flexibility, and performance of color management. This column focuses on the new features of ColorSync 2.0 and how applications can take advantage of them. (For a good review of ColorSync 1.0 and color management in general, see John Wang's Print Hints column in *develop* Issue 14.)

WHAT IS COLORSYNC 2.0?

ColorSync 2.0 is an extension to the Mac OS that provides a color management system for applications, scanner drivers, printer drivers, and other components of the OS such as QuickDraw and QuickDraw GX. The objective of the color management system is to provide consistent color across devices that have different color ranges, or *gamuts*.

All the versions of QuickDraw GX that have shipped as of this writing (v1.0.1 through v1.1.2) use the ColorSync 1.0 API. ColorSync 2.0 is backward compatible, so QuickDraw GX will work fine if ColorSync 2.0 is installed. QuickDraw GX version 1.2 will add full ColorSync 2.0 support.•

To understand the task of color management, consider the process of scanning, displaying, editing, and printing a color document: In a typical configuration, a color document may interact with three devices — scanner, monitor, and printer — each of which works with color in different ways. A scanner contains a CCD array, which is nonlinearly sensitive to specific frequencies of red, green, and blue light. A monitor hurls electrons at special phosphors to produce varying amounts of red, green, and blue light. And a color printer relies on a mixture of dyes, waxes, or toner to subtract cyan, magenta, yellow, and black from white paper. Because each of these devices uses different physical systems in different color spaces with different gamuts, providing consistent color is difficult. The goal is to provide the best consistency given the physical limitations of each device.

To meet this goal, ColorSync 2.0 requires detailed information about each device and how it represents or characterizes color. This information is encapsulated in a *device profile*. A ColorSync-savvy scanner stores (or "embeds") its profile in the document it creates. A ColorSync-savvy application uses the profile embedded in the document and displays it according to the monitor's profile; a ColorSync-savvy printer renders the document according to the printer's profile.

DEVICE PROFILES

Device profiles are the key ingredient of any color management system because they define the unique color behavior of each device. They're used by *color management module* (CMM) components, which perform the low-level calculations required to transform colors from a source device color space to a destination device color space.

CMM used to stand for *color matching method*. There was disagreement with that name because a CMM component does a lot more than just color matching. So we changed the name to *color management module* to be more accurate.•

ICC profile format. ColorSync 2.0 uses a new profile format defined by the International Color Consortium (ICC), the founding members of which include Apple, Adobe Systems, Agfa-Gevaert, Eastman-Kodak, Microsoft, Silicon Graphics, Sun, and FOGRA (honorary). The *International Color Consortium Profile Format Specification* states the following in its introduction:

> *The intent of this format is to provide a cross-platform device profile format. Such device profiles can be used to translate color data created on one device into another device's native color space. The acceptance of this format by operating system vendors allows end users to transparently move profiles and images with embedded*

DAVID HAYWARD (AppleLink HAYWARD.D) has been working in the Printing, Imaging, and Graphics group in Developer Technical Support for over a year. His proudest achievement to date is the ability to make his hour-long commute every morning without waking up until he hits the speed bumps on Apple's R&D campus. Currently Dave is developing a ColorSync CMM for his closet so that he no longer has to worry about mismatching his clothes.•

profiles between different operating systems. For example, this allows a printer manufacturer to create a single profile for multiple operating systems.

The ICC profile format is designed to be flexible and extensible so that it can be used on a wide variety of platforms and devices. The profile structure is defined as a header followed by a tag table followed by a series of tagged elements that can be accessed randomly and individually. In a valid profile, a minimal set of tags must be present, but optional and private tags may be added depending on implementation needs. Complete definitions of the required tags can be found in the profile format specification. Perhaps just as important, Apple and Adobe have defined how profiles can be embedded in the common graphics file formats PICT, EPS, and TIFF.

There have been changes in the way ColorSync works with profiles as a result of this new format. For example, with ColorSync 1.0, the entire profile format was compact enough to be used as a memory-based data structure, whereas with ColorSync 2.0, profiles can be much larger and typically are disk-based. However, ColorSync 2.0 can still make use of old 1.0 profiles for backward compatibility.

Profile types. There are three main types of device profile: input, display, and output. These types have the following signatures:

- 'scnr' — input devices such as scanners or digital cameras
- 'mntr' — display devices such as monitors or liquid crystal displays
- 'prtr' — output devices such as printers

In addition to these basic types, three other device profile types are defined:

- 'link' — Device link profiles concatenate into one profile a series of profiles that are commonly used together. A profile of this type can simplify and expedite the processing of batch files when the same combination of device profiles and non-device profiles is used repeatedly.
- 'spac' — Color space conversion profiles are used by CMMs to perform intermediate conversions between different device-independent color spaces.
- 'abst' — Abstract profiles provide a generic method for users to make subjective color changes to images or graphic objects by transforming the color data.

Profile quality and rendering intent. Typically you can think of a profile as a self-contained set of data that contains all the information needed for a CMM to perform a color match. Therefore, if an application wants to embed a profile in a document, it shouldn't have to make any changes to the profile — the profile is just a black box of data. This is true for the most part, but there are a few attributes of a profile that an application can change to modify the behavior of the profile. So, it's better to conceptualize a profile as a black box of data with a few switches on the outside. Before embedding a profile in a document, an application can toggle any of these switches by setting the appropriate bit or bits in the profile's header. One of the switches determines the profile's quality and another specifies its rendering intent:

- The quality flag bits provide a convenient place in the profile for an application to indicate the desired quality of a color match (potentially at the expense of speed and memory) as normal, draft, or best quality. In ColorSync 2.0 these qualities do not mandate the use of one algorithm over another; they're just "recommendations" that the CMM may choose to ignore or implement as it sees fit.
- The rendering intent determines how the CMM performs the match. The possible intents are photographic matching, saturation matching, relative colormetric matching, and absolute colormetric matching.

Profile header structure: CMAppleProfileHeader. In the ColorSync 1.0 profile format, the first member of the profile header structure (CMAppleProfileHeader) is a CMHeader structure, which contains all the basic information about the profile. Similarly, the ColorSync 2.0 profile begins with a CM2Header structure. The fields of the CM2Header structure are slightly different from those in the old CMHeader, to reflect some of the improvements provided by the new ICC profile format. However, to be backward-compatible with 1.0, ColorSync 2.0 defines a union of the two header structures. Because the version field is at the same offset in both header structures, it can be used to determine the version of the profile format.

Because ColorSync 2.0 provides support for ColorSync 1.0 profiles, your application should be prepared to handle both formats. Your code should always check the version field of the header before accessing any of the other fields in the header or reading any of the profile's tags.

Profile location structure: CMProfileLocation. ColorSync 2.0 profiles are typically disk-based files, but they can also be memory-based handles or pointers. To allow this flexibility, whenever a profile location needs to be specified (as a parameter for CMOpenProfile, for

example) a CMProfileLocation structure is used. This structure contains a type flag followed by a union of an FSSpec, a handle, and a pointer.

Profile reference structure: CMProfileRef. Once a profile has been opened, a private structure is created by ColorSync to maintain the profile until it's closed. A CMProfileRef (defined as a pointer to the private structure) can be used to refer to the profile.

COLOR WORLDS

A *color world* is a reference to a private ColorSync structure that represents a unique color-matching session. Although profiles can be large, a color world is a compact representation of the mapping needed to match between profiles. Conceptually, you can think of a color world as a sort of "matrix multiplication" of two or more profiles that distills all the information contained in the profiles into a fast multidimensional lookup table. A color world can be created explicitly with low-level routines such as NCWNewColorWorld or automatically with high-level routines like NCMBeginMatching.

COLORSYNC 2.0 ROUTINES

Here I'll briefly describe the most commonly used ColorSync 2.0 routines, grouped according to purpose.

The API naming convention is as follows: Calls prefixed with "CM" are high-level color management routines, while those prefixed with "CW" are low-level routines that take a color world as an argument. An "N" before "CM" or "CW" indicates calls that are new to ColorSync 2.0, to distinguish them from the old ColorSync 1.0 calls (which are still supported for backward compatibility).•

Accessing profile files. There is a set of basic routines to work with profiles as a whole. For example, CMNewProfile, CMOpenProfile, CMCopyProfile, and CMGetSystemProfile do what you would expect from their names.

Accessing profile elements. These routines perform more specific operations on profiles and profile elements. CMValidateProfile checks whether a profile contains all the needed tags, CMGetProfileElement gets a specific tag type from a profile, and CMGetProfileHeader gets the important header information of a profile.

Embedding profiles. NCMUseProfile is a simple routine for embedding a profile into a PICT. If you need to extract a profile or embed a profile into a different file format, you can use CMFlattenProfile to embed or CMUnflattenProfile to extract.

QuickDraw-specific matching. These high-level routines provide a basic API to simplify color matching for QuickDraw drawing routines. NCMBeginMatching tells Color QuickDraw to begin matching for the current graphics device using the specified source and destination profiles. NCMUseProfileComment inserts a profile as a picture comment into an open picture. NCMDrawMatchedPicture draws a picture using color matching. CWMatchPixMap matches a PixMap using the specified color world.

Low-level matching. These low-level routines create color worlds and perform color matching. NCWNewColorWorld creates a color world using the specified source and destination profiles, while CWConcatColorWorld creates one using an array of two or more profiles. Using the specified color world, CWMatchColors matches a list of colors and CWMatchBitmap matches a generic bitmap.

Searching profile files. This set of routines allows your application to search the ColorSync™ Profiles folder for the subset of profiles that meets your needs. For example, you could search for only printer profiles and use the search result to provide a pop-up menu for the user. CMNewProfileSearch searches the ColorSync™ Profiles folder for all profile files that match the supplied CMSearchRecord. The matches aren't returned to the caller, but the number of profiles matched and a reference to the search result are returned. The search result is a CMProfileSearch structure that points to private structures maintained by ColorSync and can be accessed with a call like CMSearchGetIndProfile, which opens and returns a CMProfileRef for the *n*th member of the search result.

PostScript code generation. This set of routines allows your application or printer driver to generate PostScript™ code that can be sent to a PostScript Level 2 printer so that the actual matching calculations will be performed in the printer instead of on the user's computer. CMGetPS2ColorRendering gets a color rendering dictionary (CRD) for a specified source and destination profile. CMGetPS2ColorSpace gets a color space array (CSA) for a specified source profile.

BECOMING COLORSYNC-AWARE

At the very least, your application should respect any embedded profiles in the documents it works with. For example, if your application works with PICT files, it shouldn't do anything that would strip out the ColorSync picture comments used for embedding. Even though your application may choose not to make use of the profiles, another application or printer driver may be able to take advantage of them.

PRINTING WITH COLORSYNC

If your application prints QuickDraw data to a ColorSync-savvy printer driver, you need do nothing to get matched output. When the stream of QuickDraw data sent to the driver contains an embedded profile in picture comments, the ColorSync-savvy printer driver will create a new color world to match from the embedded profile to the printer's profile. The driver will then match subsequent QuickDraw operations accordingly before sending them to the printer. If the QuickDraw data stream doesn't contain embedded profiles, the driver will use the current system profile (the profile that the user selected in the ColorSync control panel) as the source profile. That way, the printed output will match the screen display.

One example of a ColorSync-savvy printer driver is the LaserWriter 8.3 driver. Whereas previous versions of LaserWriter 8 allowed the user to choose between "Black and White" and "Color/Grayscale" in the Print dialog, this version adds two new choices. "ColorSync Color Matching" tells the driver to use ColorSync to match an image on the host Macintosh before sending it to the printer. The other option, "PostScript Color Matching," instructs the driver to generate PostScript CSAs and CRDs, which are sent to the printer so that the actual matching is performed in the printer. (The ColorSync API is used to generate the CSAs and CRDs according to the source profiles that may be embedded in the document and the destination profile of the printer.) In either case, the LaserWriter 8.3 driver allows the user to choose a printer profile from a list of printer profiles installed in the ColorSync™ Profiles folder.

Because ColorSync-savvy printer drivers do much of the work for you, it's best if your application prints documents with QuickDraw even if they're not PICT files. For example, if your application reads and prints TIFF files, the best approach is to convert the TIFF data (which may have a profile embedded in tags) to a PicHandle (which would have the profile embedded in picture comments). To print, you draw the PicHandle with DrawPicture into the printer's color graphics port.

If the printer's driver doesn't support ColorSync, your application can still use ColorSync to produce matched output as long as you have an appropriate profile for the device. (There are several commercial tools that build ICC profiles.) Given a source and destination profile, you can use the ColorSync API to match the image or, if your application must send PostScript data directly to a printer, to generate CRDs.

WHAT ELSE A COLORSYNC-SAVVY APPLICATION CAN DO

There is much that an application can do with ColorSync that will help the user work with color. For starters, an application could do the following:

- Provide the user with information on any profiles embedded in a document, and possibly also allow the user to change the quality and rendering intent settings of embedded profiles.
- Include a print preview mode that shows a "soft proof" of the matched output on the display. The application accomplishes this by building a color world with CWConcatColorWorld that matches through three profiles: from the source profile (which is embedded in the document) to the printer's profile (which you allow the user to pick from a list of installed printer profiles) and back to the screen profile (which is the current system profile).
- Along with soft-proofing, it's useful to show the user what colors in the document are out of gamut according to the current destination profile. Gamut checking can be done with routines such as CWCheckColors and CWCheckBitmap.

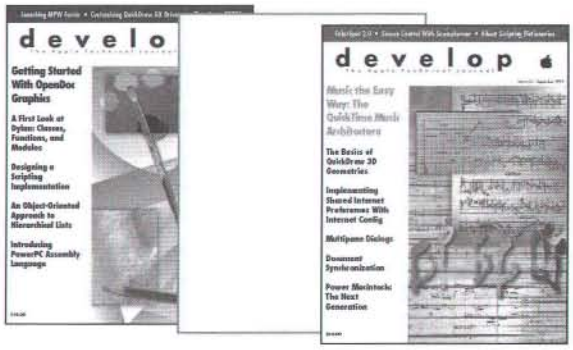
Note that the LaserWriter engineering team is designing new PrGeneral code for the 8.3.1 version of the driver. This will allow an application to determine what profile is selected in the Print dialog.

WHERE TO GO FOR MORE

Everything you need to use ColorSync 2.0, including interfaces, libraries, sample code, utilities, and the ICC profile format specification, is on this issue's CD and in the Mac OS Software Developer's Kit. The technical reference for ColorSync 2.0 consists of several chapters in the book *Advanced Color Imaging on the Macintosh*, which is also included on this issue's CD and will soon be available in print from Addison-Wesley; this documentation covers everything from a high-level discussion of color management theory to a detailed description of the ColorSync 2.0 API. Why not take a closer look and see how you can take of advantage of this new improved technology in your application?

Thanks to Paul Danbold, Steve Swen, Nick Thompson, and John Wang for reviewing this column.•

Missing something?



Are there issues of *develop* that have passed you by? If you'd like to complete your *develop* collection, full-color, bound copies are available for $13 per issue, including shipping and handling. (Back issues are also on the *develop Bookmark* CD and the *Developer CD Series* Reference Library edition, as well as on AppleLink, eWorld, and the Internet.) For more information about how to order printed back issues (and where to find them online), see the inside front cover of this issue. *Supplies are limited. Please allow 4 to 6 weeks for delivery.*

Issue 1 Color; Palette Manager; Offscreen Worlds; PostScript; System 7; Debugging Declaration ROMs

Issue 2 C++ (Objects; Style Guide); Object Pascal; Memory Manager; MacApp; Object-Based Design

Issue 3 ISO 9660 and High Sierra; Accessing CD Audio Tracks; Comm Toolbox; 8•24 GC Card; PrGeneral

Issue 4 Device Driver in C++; Polymorphism in C++; A/ROSE; PostScript; Apple IIGS Printer Driver

Issue 5 (Volume 2, Issue 1) Asynchronous Background Networking; Palette Manager; Macintosh Common Lisp

Issue 6 Threads; CopyBits; MacTCP Cookbook

Issue 7 QuickTime 1.0; TrueType; Threads and Futures; C++ Objects in a World of Exceptions

Issue 8 Curves in QuickDraw; Date and Time Entry in MacApp; Debugging; Hybrid Applications for A/UX

Issue 9 Color on 1-Bit Devices; The TextBox You've Always Wanted; Sound; Text Windows via the Terminal Manager; Tracks: A Tool for Debugging Drivers

Issue 10 Apple Event Objects; Enhancements for the LaserWriter Font Utility; GWorlds; The Optimal Palette

Issue 11 Asynchronous Sound; Multibuffering Sounds; Exceptions; NetWork: Distributed Computing

Issue 12 Components; Time Bases; Apple Event Coding Through Objects; Globals in Standalone Code

Issue 13 QuickTime and Component-Based Managers; Asynchronous Routines; Macintosh Debugging Revisited; Adventures in Color Printing; DeviceLoop

Issue 14 Writing Localizable Applications; 3-D Rotation Using a 2-D Input Device; Video Digitizing Under QuickTime; Making Better QuickTime Movies

Issue 15 QuickDraw GX; Component Registration; Floating Windows; Working in the Third Dimension

Issue 16 Making the Leap to PowerPC; PowerTalk; Drag and Drop From the Finder; Color Matching With QuickDraw GX; International Number Formatting

Issue 17 Proto Templates on the Newton; Standalone Code on PowerPC; Debugging on PowerPC; Thread Manager; Window Zooming

Issue 18 Apple Guide; Open Scripting Architecture; Graphics Speed on the Power Macintosh; Displaying Hierarchical Lists; Preferences Files

Issue 19 OpenDoc Part Handlers; PowerPC Memory Usage; Designing for the Power Macintosh; QuickDraw GX (Printing; Bitmaps); Inheritance in Scripts

Issue 20 AOCE; Make Your Own Sound Components; Scripting the Finder; NetWare on PowerPC

Issue 21 OpenDoc Graphics; Dylan; Designing a Scripting Implementation; Object-Oriented Hierarchical Lists; Introducing PowerPC Assembly Language

Issue 22 QuickDraw 3D; Copland; PCI Device Drivers; Custom Color Search Procedures; The OpenDoc User Experience; Futures

The Basics of QuickDraw 3D Geometries

No matter how realistic or sophisticated you want your 3D images to be, you must always build objects with the primitive geometric shapes provided by the graphics system. Our article in Issue 22 gave the basic information you need to start developing applications with QuickDraw 3D. Here we delve deeper into the primitive geometric shapes provided by QuickDraw 3D and show how to use them effectively. We also give you some tips we've gained from working with developers.

NICK THOMPSON AND PABLO FERNICOLA

Geometric shapes — or geometries — form the foundation of any 3D scene. QuickDraw 3D provides a rich set of primitive geometric types that you use to define the shapes of things. You can apply attributes (such as colors) to geometric objects, collect geometric objects into groups, and copy, illuminate, texture, transform, or otherwise modify them to attain the visual effects you want. In other words, everything that's drawn by QuickDraw 3D is either a geometry or a modification of a geometry. So you need to know how to define geometries (and usually also how to create and dispose of them) to work effectively with QuickDraw 3D. This article describes the geometries available in QuickDraw 3D version 1.0 and shows how they relate to other aspects of the QuickDraw 3D architecture (such as the class hierarchy).

We're assuming that you're already familiar with the basic capabilities of QuickDraw 3D. For a good introduction, see our article "QuickDraw 3D: A New Dimension for Macintosh Graphics" in Issue 22 of *develop* (a copy is on this issue's CD). In that article, we provided an overview of QuickDraw 3D's architecture and capabilities. You can think of QuickDraw 3D as having three main parts: graphics, I/O (the QuickDraw 3D metafile), and human interface guidelines. Here, we provide more detail on the graphics portion of the QuickDraw 3D API and highlight some parts of that API that could use clarification as you try to implement geometries.

NICK THOMPSON (AppleLink NICKT) from Apple's Developer Technical Support group took a trip to Las Vegas this year in a rented Cadillac. He was impressed by some of the ancient architecture on show in this fine city, such as the Pyramid of Luxor, Excalibur's Castle, and Caesar's Palace (he was surprised that the ancient Egyptians, King Arthur, and the Roman emperor had all made it that far west). He was also impressed by the free food and drinks — all he had to do was sit at a table and buy small plastic disks with green scraps of paper that he got from a hole in the wall. Having rented a Cadillac for this trip, Nick now has his heart set on a 1968 Eldorado convertible.

PABLO FERNICOLA (AppleLink PFF, eWorld EscherDude), the short one in the picture, is the brains behind the operation. His hobbies include traveling to exotic places (such as the local supermarket), eating fine cuisine, and talking to his dog (who is almost as big as Nick, and probably a lot smarter). He's hard at work on the next generation of QuickDraw 3D, which — like Pablo — is bound to be even smarter. Pablo says, "You can use QuickDraw 3D's metafile format everywhere, even for defining virtual environments on the net. So get those applications ready, won't you?"

To help you get started using geometries, this issue's CD contains version 1.0 of the QuickDraw 3D shared library and programming interfaces, sample code, and an electronic version of the book *3D Graphics Programming With QuickDraw 3D*, which provides complete documentation for the QuickDraw 3D programming interfaces.

A WORD ABOUT RENDERING AND SUBMITTING

Our previous article included an introduction to rendering; we'll review a key concept here — retained vs. immediate rendering. We'll also elaborate on an important point we glossed over in that article: submitting something to be rendered rather than just rendering it. These concepts will help set the stage for what you'll learn here about working with geometries.

RETAINED VS. IMMEDIATE MODE RENDERING

A powerful feature of QuickDraw 3D is that it supports both retained and immediate modes for rendering geometric data; you can even mix these modes within the same rendering loop. In *retained mode*, the definition and storage of the geometric data are kept internal to QuickDraw 3D — as abstract geometric objects. In *immediate mode*, the application keeps the only copy of the geometric data; for efficiency, the application should use QuickDraw 3D data structures to hold the data, but those structures can be embedded in application-defined structures. Retained mode geometric objects and immediate mode geometric data define the shapes of objects. You'll typically use one or more primitive geometric types provided by QuickDraw 3D (such as triangles or meshes) to build up a scene.

Whether you use retained or immediate mode to render geometries usually depends on how much of a model changes from one rendering operation to the next. As we'll illustrate with examples in this section, we prefer to use retained geometries most of the time and to use immediate mode only for temporary objects. Since our preference for retained mode is a departure from the traditional QuickDraw way of drawing, we'll attempt to convince you that retained mode is a much more efficient method of rendering geometries.

Immediate mode. When you use immediate mode rendering, the data that defines a geometry is stored and managed by your application. For example, to draw a triangle you would write code similar to that in Listing 1. If you wanted to draw this triangle many times, or from different camera angles, you would have to maintain the data in your application's data structures.

Typically when using immediate mode, you stick to a single type of geometry (triangles are popular with developers accustomed to lower-level 3D graphics

Listing 1. Rendering a triangle in immediate mode

```
TQ3TriangleData   myTriangle;

// Set up the triangle with appropriate data.
...
// Render the triangle.
Q3View_StartRendering(myView);
do {
   Q3Triangle_Submit(&myTriangle, myView);
} while (Q3View_EndRendering(myView) == kQ3ViewStatusRetraverse);
```

libraries). If you use multiple geometric types, you need to define a data structure to manage the order of the geometries. An example of rendering several geometries in immediate mode is shown in Listing 2.

Listing 2. Rendering several geometries in immediate mode

```
typedef struct myGeometryStructure {
   TQ3ObjectType                type;
   void                         *geom;
   struct myGeometryStructure   *next;
} myGeometryStructure;

myGeometryStructure     *currentGeometry;
...
Q3View_StartRendering(myView);
do {
   while (currentGeometry != NULL) {
      switch (currentGeometry->type) {
         case kQ3GeometryTypeTriangle:
            Q3Triangle_Submit((TQ3TriangleData *) currentGeometry->geom,
               myView);
         break;
         case kQ3GeometryTypePolygon:
            Q3Polygon_Submit((TQ3PolygonData *) currentGeometry->geom,
               myView);
         break;
      }
      currentGeometry = currentGeometry->next;
   }
} while (Q3View_EndRendering(myView) == kQ3ViewStatusRetraverse);
```

If you wanted to apply transforms to a geometry as it's being drawn, you would have to add a new case to the switch statement. This gets complicated pretty quickly. As a result, many developers, when given a choice, will use immediate mode only for models that have a fixed geometry and are not being altered.

Retained mode. Creating geometric objects allows renderers to take advantage of characteristics of particular geometries and thus optimize the drawing code. The code in Listing 3 draws a triangle in retained mode.

SUBMITTING

You'll notice that the routine to draw an object is Q3Object_Submit. This probably seems a bit strange: why didn't we call it Q3Object_Draw? The reason is that there are four occasions in which you need to specify a geometry — when writing data to a file, when picking, when determining the bounds of a geometry, and when rendering — and QuickDraw 3D provides a single routine that you use in all of these cases. To indicate which operation you want to perform, you call the Submit routine inside a loop that begins and ends with the appropriate calls. For instance, to render a model, you call Submit functions inside a rendering loop, which begins with a call to Q3View_StartRendering and ends with a call to Q3View_EndRendering (as shown in Listing 3). Similarly, to write a model to a file, you call Submit functions inside a writing loop, which begins with a call to Q3View_StartWriting and ends with a call to Q3View_EndWriting.

Listing 3. Rendering a triangle in retained mode

```
TQ3TriangleData   triangleData;

// Set up the triangle with appropriate data.
...
// Create the triangle.
triangleObject = Q3Triangle_New(&triangleData);
// Render the triangle.
Q3View_StartRendering(myView);
do {
   Q3Object_Submit(triangleObject, myView);
} while (Q3View_EndRendering(myView) == kQ3ViewStatusRetraverse);
```

Listing 4. A submitting function

```
// Submit the scene for rendering, file I/O, bounding, or picking.
TQ3Status SubmitScene(DocumentHdl theDocument)
{
   TQ3Vector3D    globalScale, globalTranslate;

   globalScale.x = globalScale.y = globalScale.z =
                                    (**theDocument).fGroupScale;
   globalTranslate = *(TQ3Vector3D *)&(**theDocument).fGroupCenter;
   Q3Vector3D_Scale(&globalTranslate, -1, &globalTranslate);
   Q3Style_Submit((**theDocument).fInterpolation,
      (**theDocument).fView);
   Q3Style_Submit((**theDocument).fBackFacing, (**theDocument).fView);
   Q3Style_Submit((**theDocument).fFillStyle, (**theDocument).fView);

   Q3MatrixTransform_Submit(&(**theDocument).fRotation,
      (**theDocument).fView);
   Q3ScaleTransform_Submit(&globalScale, (**theDocument).fView);
   Q3TranslateTransform_Submit(&globalTranslate, (**theDocument).fView);
   Q3DisplayGroup_Submit((**theDocument).fModel, (**theDocument).fView);

   return (kQ3Success);
}
```

We recommend that you put all your Submit calls together within a single function (such as the one shown in Listing 4) that you can then call from your rendering loop, picking loop, writing loop, or bounding loop. Organizing your code in this fashion will prevent a common mistake: creating rendering loops that are out of sync with picking or bounding loops. It also simplifies your rendering and picking loops — you just call your submitting function from within the loop. Here's an example of calling the function in Listing 4 from within a rendering loop:

```
Q3View_StartRendering((**theDocument).fView);
do {
   theStatus = SubmitScene(theDocument);
} while (Q3View_EndRendering((**theDocument).fView) ==
                                          kQ3ViewStatusRetraverse);
```

QUICKDRAW 3D CLASS HIERARCHY

Even if you perform all your rendering in immediate mode — that is, without creating any QuickDraw 3D geometric objects — you still need to create some QuickDraw 3D objects, such as a view, camera, and draw context, in order to render any image at all. So working with geometries in QuickDraw 3D means working with at least some objects. Before going into detail about how to create and use QuickDraw 3D geometric objects, let's review the object system and some of its basic classes.

QuickDraw 3D is an object-based system. We chose to implement the API with the C language, which doesn't support objects directly; nevertheless QuickDraw 3D is organized into a definite class hierarchy. Figure 1 shows part of this hierarchy, emphasizing the classes that are discussed in this article. At the top of the class hierarchy is the basic QuickDraw 3D Object class. Geometries, such as the triangle, polygon, and mesh classes, are at the bottom of the hierarchy.

The Object class is really named TQ3Object. This article uses shortened forms of the QuickDraw 3D class names.•

You can determine the class in which a function is defined simply by looking at the function's name: function names have the form *Q3class-name_method*. For example, the function Q3Shared_GetReference is defined in the Shared class and returns a reference to the shared object that's passed as an argument. The function Q3Object_Dispose is defined in the Object class; it accepts any QuickDraw 3D object as an argument (since Object is the root class) and disposes of it.

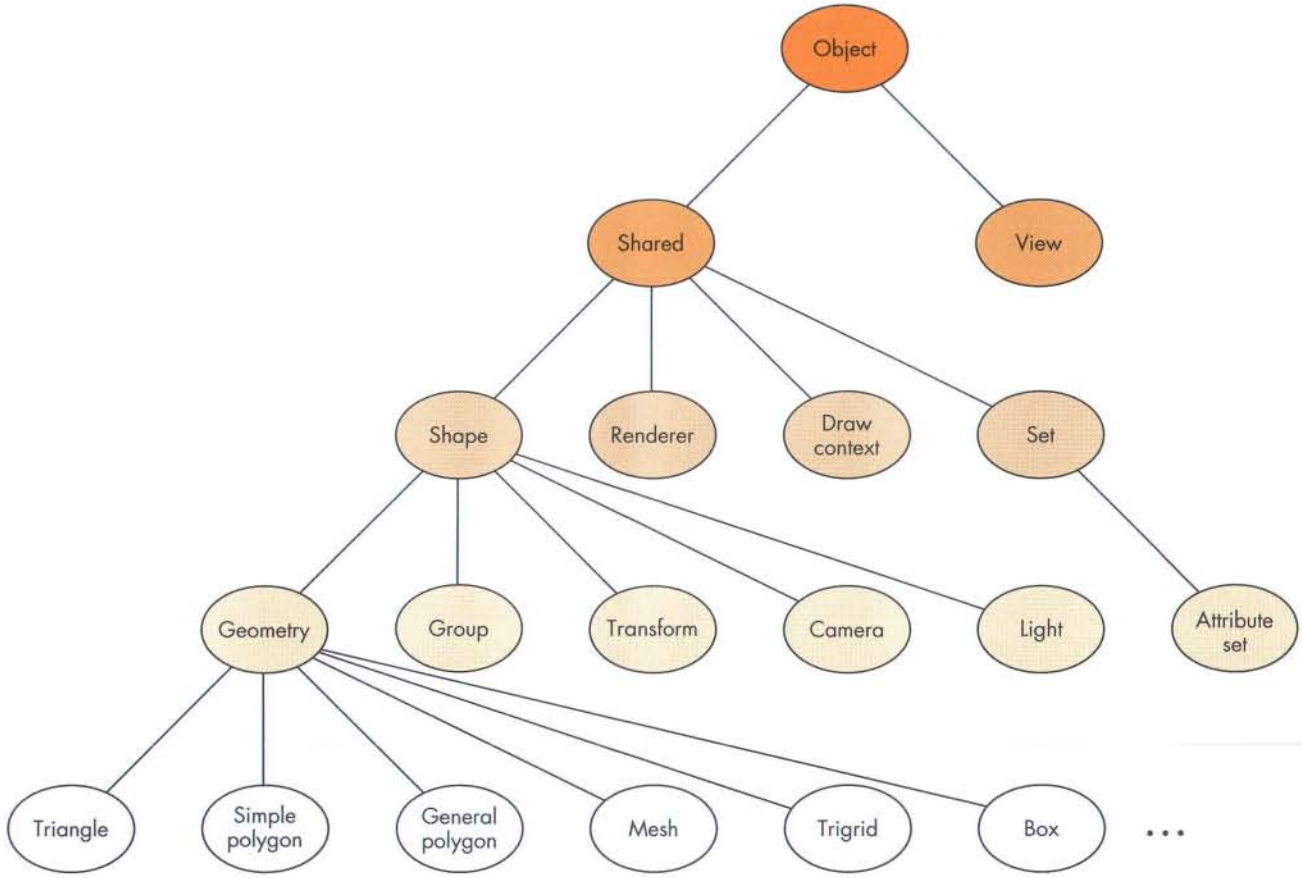


Figure 1. Partial QuickDraw 3D class hierarchy

In the following sections, we'll talk more about the classes shown in Figure 1 and answer some questions developers have had about using them when working with geometries. Then we'll (finally!) talk about the geometric objects themselves and provide sample code for using many of them.

THE SHARED CLASS

Generally speaking, drawing anything with QuickDraw 3D involves working with objects that inherit from the Shared class. There can be multiple references to shared objects (hence the name); the way QuickDraw 3D determines whether a shared object is still referenced is by way of a *reference count*, initially 1. Developers new to QuickDraw 3D are sometimes confused by reference counts, but they're actually very straightforward. When you create a shared object, its reference count is 1. For example:

```
myNewObject = Q3Mesh_New();
// myNewObject now has a reference count of 1.
```

When you get a shared object as a result of a Get call, or pass one as an argument in an Add or Set call, the object's reference count is incremented.

```
// The following calls increment the object's reference count.
Q3Group_GetPositionObject(myGroup, currentPosition, &myExistingObject);
...
Q3Group_AddObject(myGroup, myObject);
...
Q3View_SetDrawContext(myView, myDrawContext);
```

Passing a shared object as the argument to a Dispose call decrements its reference count; only when the count goes to 0 does QuickDraw 3D actually dispose of the memory occupied by the object. As a general rule, you should dispose of the object before the scope of the variable expires. For example:

```
{  // Start of the block. Variables come into scope.
   TQ3Object   myObject = Q3Mesh_New();   // The start of myObject's scope

   // Do something that manipulates myObject.
   ...
   // The scope of myObject is going to end at the next closing brace,
   // so dispose of it before we go out of scope.
   Q3Object_Dispose(myObject);
}  // End of the block.
```

If you were assigning an object reference to a global variable, you would dispose of the object before calling Q3Exit and exiting your program.

Q: Why does my application crash when I call Q3Exit?

A: In the debugging version of QuickDraw 3D, Q3Exit generates a debugging message for each remaining object. The default behavior is to display the message with the DebugStr call; the message is displayed in MacsBug (or whatever debugger you use). So your application isn't crashing; it's trying to tell you to tidy up after yourself! To avoid this unscheduled trip into your debugger, you can install your own error handler and log the message to a file. And, of course, you should fix your application so that it doesn't leak memory!•

Let's take a closer look at what happens to reference counts when you create and dispose of a shared object. Figure 2 shows the typical lifetime of a group of QuickDraw 3D objects (we'll talk more about groups later).

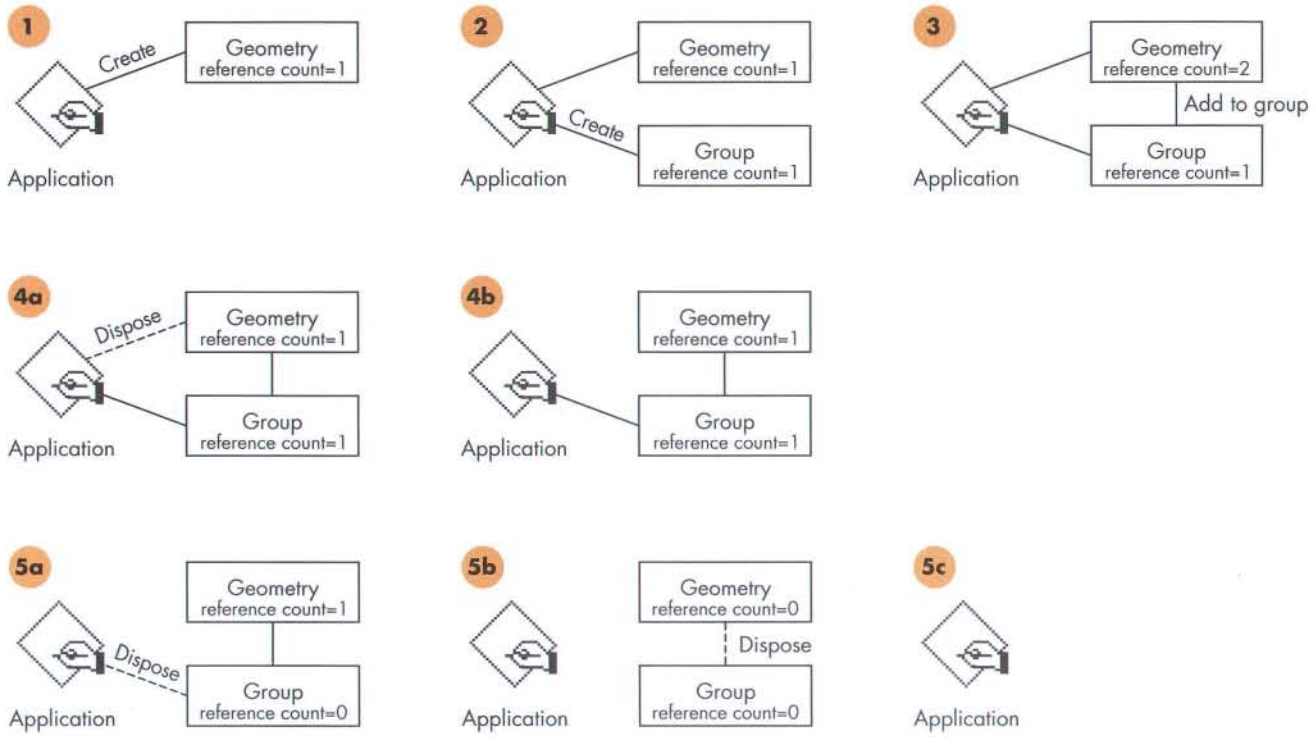


Figure 2. Reference counts in QuickDraw 3D

1. An application creates a geometric object. Its reference count is 1.
2. The application creates a group object. Its reference count is also 1.
3. The application adds the geometry to the group (by calling the function Q3Group_AddObject), which increments the reference count of the geometric object (to 2).
4. The application disposes of the geometric object (by calling the function Q3Object_Dispose), which is safe to do once it's added to the group. This decrements the reference count of the geometry back to 1. The application can then operate on the group (which now contains the geometry).
5. When it's finished with the group, the application can dispose of the group object. This lowers the reference count of the group to 0, which causes QuickDraw 3D to dispose of the group and of all the objects within the group. As you can see, the geometry is disposed of as a side effect of disposing of the group.

THE VIEW CLASS

The view object ties together the elements required to draw a scene; it's the central object that holds the state information for rendering a scene. A *scene* consists of the geometry being drawn (hereafter referred to as the *model*), together with the light, camera, draw context, and other objects. Our previous article discussed how to set up a view; we'll expand on that discussion by describing how to create and manage multiple scenes of a model.

To display a scene, you need at least one view object, and each view object must have a camera associated with it. Each of your application's windows usually has one view object attached to it. When you need to display multiple scenes of the same model, you can create multiple windows, each with its own view object. Then you simply

submit the model to the desired view. Alternatively, you can display multiple scenes using a single view object by setting up several different cameras and draw contexts and switching between them — manipulating the view's camera to create each scene (see Figure 3).

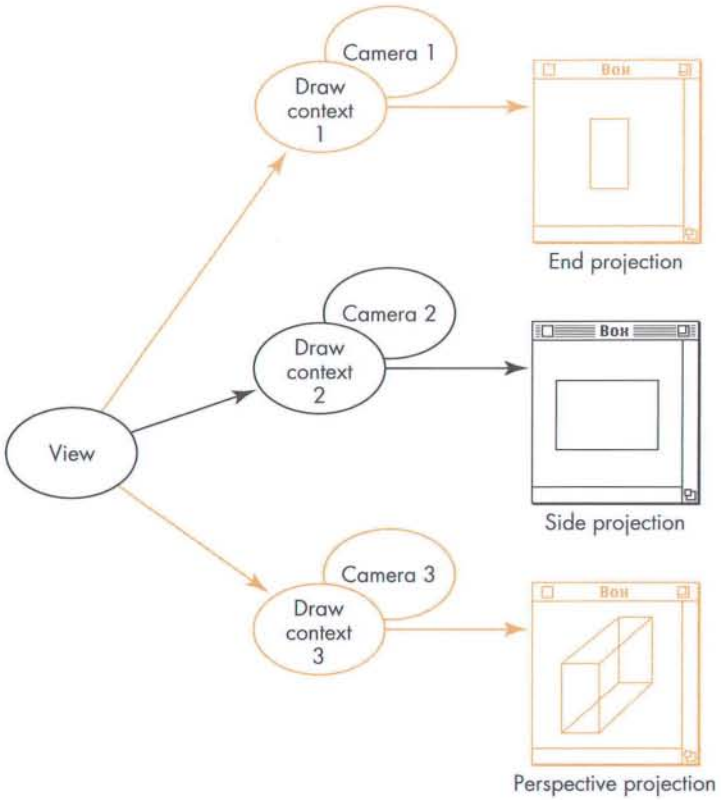


Figure 3. Multiple scenes of the same model

You can have only one active draw context and camera for each view object, so to update one of your windows, you need to manually set the active draw context and camera for the appropriate scene. For this reason, the first option (one view per window) is usually simpler to implement.

THE GROUP CLASS

QuickDraw 3D provides a number of classes for grouping objects together. Groups are useful because they provide a structure to your models, allowing you to express the relationship between different geometric objects. Of course, if you want to use your own data structures for storing your geometries, you can do so, but generally it's more work. Using QuickDraw 3D's group classes, you can create hierarchies of geometric data by nesting groups within other groups. Figure 4 shows the group classes provided with QuickDraw 3D.

You can create a group object by calling Q3Group_New. This creates an object belonging to the generic Group class. QuickDraw 3D provides the following subgroups of the generic Group class, which are distinguished by the types of objects you're allowed to place in them:

- A *light group* places the light objects for a scene in a group, which simplifies lighting management. For example, you could provide an iterator function to loop through the group and turn all the lights on or off.

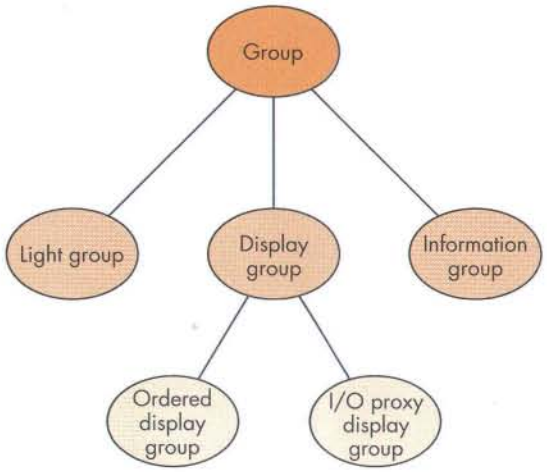


Figure 4. Group classes provided by QuickDraw 3D

- A *display group* manages objects that are drawable, including geometries, styles, and transforms. You can use the function Q3Object_IsDrawable to confirm whether an object is drawable.
- An *information group* stores informational strings, such as the author, copyright, trademark, and other human-readable information within a metafile.

Because we want to talk about geometries, which are drawable objects, we'll concentrate on display group objects. In addition to "plain" display groups, there are two specialized subclasses of the display group class: ordered and I/O proxy. For a plain display group, the order in which items are placed in the group is the order in which they're drawn when the group is submitted, regardless of the class that the objects belong to. For an ordered display group, objects in the group are sorted by object type and are submitted (when you call Q3DisplayGroup_Submit) in the following order: transforms, styles, attribute sets, shaders, geometric objects, groups.

Ordered display groups are most useful when you want to operate on a group of objects as a single entity. For example, you know that transforms are always at the start of the group, so you could manipulate the transform to alter the orientation of the entire group. (If you were using a plain display group, you would have to search for the transform, or otherwise store a reference to it, which makes life more complicated.) Sometimes you'll want to nest a number of ordered display groups within a plain display group. If you were animating a robotic arm, for example, each component of the arm could be an ordered display group that's nested within a plain display group.

You can use I/O proxy display groups to provide multiple representations of the same data. This is useful when dealing with applications that aren't based on QuickDraw 3D or that run on other platforms. For example, some applications might be able to read only mesh objects; your application may want to use NURB patches (another type of geometric object), but you want other applications to be able to read your metafiles. In this case, you could write a NURB patch representation of your data, followed by a mesh representation. To provide both representations of the same data in a metafile, you would create an I/O proxy group that contains the NURB patch object first and the mesh object second, and write the group to the metafile. When you draw with QuickDraw 3D, the objects that appear first in the group are preferred over later objects in the group.

THE TRANSFORM CLASS

The Transform class enables you to change the position, orientation, or size of geometries. When you specify the coordinates for the vertices that define a geometry, the *x*, *y*, *z* values are expressed as floating-point values in *local coordinates*. Rendering, however, and associated operations like backface removal and lighting are performed in *world coordinates*. To transform a geometry from one space to another, QuickDraw 3D multiplies the local coordinates by a local-to-world matrix. The default value for this matrix is the identity matrix, which leaves the original geometry unchanged. By changing the value of the local-to-world matrix, you can transform geometries without having to change the geometries' coordinates.

Using an example from our previous article, let's say that you have a model that contains several boxes (see Figure 5). We could enter the coordinates for the points that make up each of the four boxes, but that's a lot of work (and if you're creating an object for each box, it's a waste of memory). Instead, we define one box at a certain location and call it the reference box. To get the effect of four boxes in different locations, we draw the reference box four times — changing the local-to-world matrix each time before drawing.

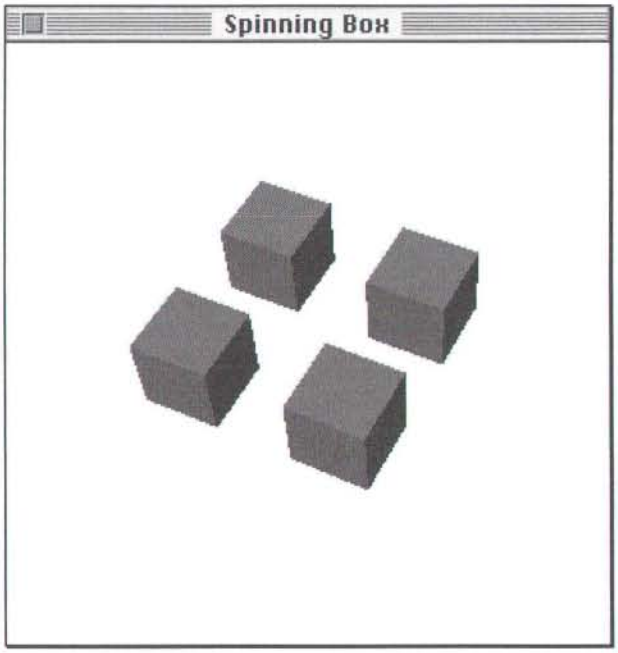


Figure 5. Boxes drawn by changing the local-to-world matrix four times

If you look in the file QD3DTransform.h, you'll notice that there are several different types of transforms. The most general type is the matrix transform, which is a 4 x 4 matrix. To use this transform, you supply the translation, rotation, and scale values in the appropriate entries of the matrix, as shown in Figure 6. You can do any type of transform that can be expressed as a 4 x 4 matrix. In the figure, you can see that the upper 3 x 3 submatrix is a rotation matrix, with the entries in the main diagonal containing the scale factors for x, y, and z. The lower row contains the translation factors.

If you know which type of transform you'll be applying, however, it's better to use one of the more specific types. In this way, QuickDraw 3D renderers and shaders will be able to take advantage of the information contained in the transform; for example, if your local-to-world matrix is just a translate transform, the renderer

$$\begin{bmatrix} S_x * R_{0,0} & R_{0,1} & R_{0,2} & 0.0 \\ R_{1,0} & S_y * R_{1,1} & R_{1,2} & 0.0 \\ R_{2,0} & R_{2,1} & S_z * R_{2,2} & 0.0 \\ T_x & T_y & T_z & 1.0 \end{bmatrix}$$

Note: S is the scale transform, R is the rotate transform, and T is the translate transform.

Figure 6. A matrix transform

doesn't have to transform normals before performing the backface removal operation (because directions are not affected by translations). Also, using the more specific types provides a better abstraction and tends to make the logic of your code easier to understand (and you don't have to deal with all those pesky matrices).

When you change the local-to-world matrix by applying transforms, each transform is concatenated as it's applied through a Submit call. For example, if before drawing a point object, we submit a translate transform, a rotate transform, a scale transform, and then a point, the point will be transformed as follows:

$$p' = p * S * R * T$$

p' is the resulting transformed point and p is the original point. T is the matrix containing the translate operation, R is the matrix containing the rotate operation, and S is the matrix containing the scale operation.

You can apply transforms either by using immediate mode calls or by creating transform objects — just as you do for geometries. Note that transforms accumulate; that is, if you apply a translation, any objects drawn after that will be translated by the same amount. If you want a transform to apply to a certain object only, you can use the Q3Push_Submit and Q3Pop_Submit calls around it or place the object in a group, since groups perform an implicit push and pop (you can change this behavior if you want).

So, let's build on what we've learned so far. We want to draw the model shown in Figure 5. Let's first do it by submitting new transforms in immediate mode, before each box is drawn, as shown in Listing 5.

Alternatively, we could create the model of the four boxes as a group, as shown in Listing 6.

THE ATTRIBUTE SET CLASS

Attributes affect the way an object is rendered in QuickDraw 3D. A view has a default set of attributes, defined in the QD3DView.h file, that can be modified to suit a particular application. If no attributes are supplied for the objects being rendered within a view, the default view attributes are applied. Attributes can be applied in a number of ways: by submitting them to a view object; by adding them to a group; or by attaching them to a geometry, to a geometry's face, or to each vertex of a geometry.

The order in which attribute sets are applied during rendering is based on a fixed hierarchy, as illustrated in Figure 7. Attributes of the same type (such as diffuse color) can override one another; they use the following preference hierarchy, from highest to lowest precedence: vertex, face, geometry, group, view. For example, a specular color attribute at the vertex level does not override a diffuse color attribute at the geometry level, whereas a specular color attribute at the vertex level does override a

Listing 5. Using translate transforms in immediate mode

```
Q3View_StartRendering(viewObject);
do {
   TQ3Vector3D translationX = {2.0,  0.0, 0.0},
               translationY = {0.0, -2.0, 0.0};

   Q3View_Push(viewObject);

   // Note how we are using a retained mode geometry with immediate mode
   // transforms. As we execute each of the calls, the boxes are drawn.

   Q3Object_Submit(referenceBox, viewObject);
   // Move to the right.
   Q3TranslateTransform_Submit(&translationX, viewObject);
   Q3Object_Submit(referenceBox, viewObject);
   // The Pop will move back to the left.
   Q3View_Pop(viewObject);
   // Move down.
   Q3TranslateTransform_Submit(&translationY, viewObject);
   Q3Object_Submit(referenceBox, viewObject);
   // Move to the right.
   Q3TranslateTransform_Submit(&translationX, viewObject);
   Q3Object_Submit(referenceBox, viewObject);
} while (Q3View_EndRendering(viewObject) == kQ3ViewStatusRetraverse);
```

Listing 6. Creating translate transform objects

```
TQ3GroupObject       myModel;
TQ3Vector3D          translationX = {2.0, 0.0, 0.0},
                     translationYAndNegativeX = {-2.0, -2.0, 0.0};
TQ3TransformObject   xform_x, xform_yx;

// Note that as we execute these calls, nothing is drawn.

myModel = Q3Group_New();
xform_x = Q3TranslateTransform_New(&translationX);
xform_yx = Q3TranslateTransform_New(&translationYAndNegativeX);
Q3Group_AddObject(myModel, referenceBox);
Q3Group_AddObject(myModel, xform_x);
Q3Group_AddObject(myModel, referenceBox);
Q3Group_AddObject(myModel, xform_yx);
Q3Group_AddObject(myModel, referenceBox);
Q3Group_AddObject(myModel, xform_x);
Q3Group_AddObject(myModel, referenceBox);

// To draw the boxes, you would call Q3Object_Submit(myModel, myView)
// within a submitting loop.
```

specular color attribute at the geometry level (because they are attributes of the same type). If attributes at any level are not supplied, the parent's attributes apply. If there are no attributes supplied anywhere in the hierarchy, the default attribute set for the view will be used.

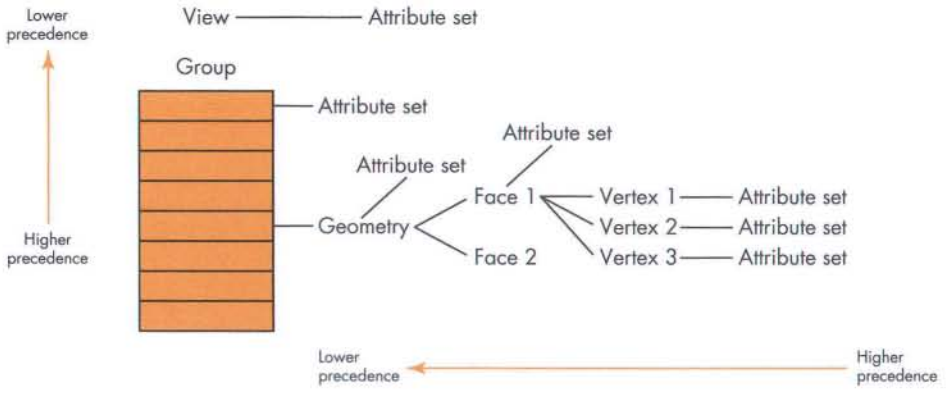


Figure 7. Hierarchy of applying attributes to a geometry

Here are the six most commonly used predefined attribute types that you can specify (there are 12 in all):

- The *diffuse color* is the actual color of the object.
- The *specular color* is the color of the light reflected by the object, which may or may not be the same as the diffuse color.
- The *specular control* determines how much light of the specular color is reflected.
- The *ambient coefficient* determines how much the ambient lighting affects the object.
- The *surface UV* attribute specifies how a texture is mapped to a geometry's vertex.
- A *texture shader* can be applied to a surface that has UV parameterization applied (more on this later).

You can also define your own custom attributes. Later, in the geometry code samples, we'll create attribute sets to affect the way the geometries are drawn.

BUILDING GEOMETRIES

Now we're ready to look at the specific geometries and show how to build them. QuickDraw 3D version 1.0 supports 12 geometries (illustrated in Figure 8). In the code examples later in this article, we'll cover the most commonly used geometries.

- A *marker* object is a bitmap that's displayed face-on at any orientation — similar to a sprite. It's useful for denoting the position of objects and for providing annotations, such as labels on objects in a 3D chart.
- A *point* object is the most basic object in QuickDraw 3D; it specifies discrete points in a scene.
- A *line* object is a line between two points.
- A *polyline* object is a line that consists of multiple segments.
- A *triangle* object is a closed planar geometry defined by three intersecting lines. It's the simplest form of a polygon.
- A *simple polygon* object is a planar geometry described by a list of vertices; it's a figure formed by a closed chain of intersecting straight lines. A simple polygon consists of a single convex contour and may not contain holes.

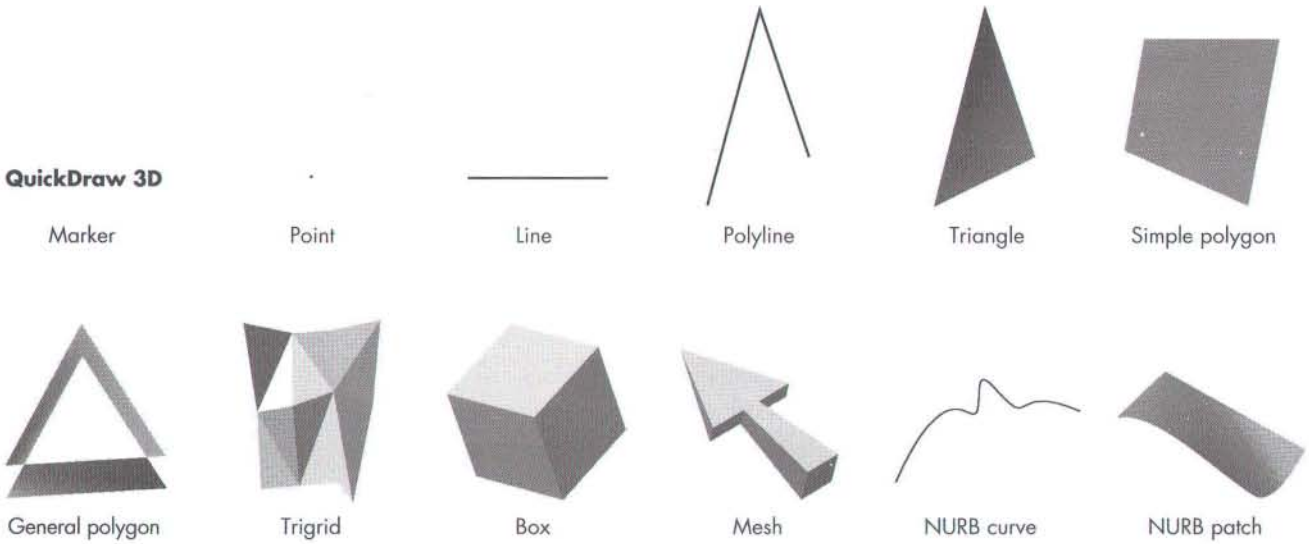


Figure 8. QuickDraw 3D geometries supplied in version 1.0

- A *general polygon* object is a planar geometry that may contain holes, be concave, and consist of one or more contours.
- A *trigrid* object is a grid whose surface consists of multiple triangles that share edges and vertices.
- A *box* object is a three-dimensional rectangular object.
- A *mesh* object is a collection of vertices, faces, and edges that represent a topological polyhedron. It's sometimes referred to as a winged-edge structure.
- A *NURB curve* object is a curve described by a NURB equation.
- A *NURB patch* object is a three-dimensional surface described by a NURB equation.

NURB stands for nonuniform rational B-spline. A B-spline is a parametric curve (a curve defined by coordinates derived from functions sharing a common parameter) whose shape is determined by a series of control points whose influence is described by basis functions.•

SIMPLE GEOMETRIES

Let's start with some simple geometries first: lines, polylines, triangles, simple polygons, and general polygons. In essence, these are the building blocks for QuickDraw 3D. You can use combinations of these to construct your model, or you can use some of the composite geometries, such as meshes and trigrids (described later).

Line and polyline objects. Lines are defined by two noncoincident points. If you want to have multiple line segments, you can use polylines (see Listing 7). In polylines, every vertex after the first one defines a new line. You can attach attributes at the geometry level or at the vertex level (which is useful for having multicolored lines, but remember that you need to use per-vertex interpolation when rendering in order for the multiple colors to apply).

Triangle objects. Triangles are the most basic of the planar geometries in QuickDraw 3D. Triangles are defined by three noncolinear, noncoincident vertices.

Listing 7. Creating a polyline

```
TQ3ColorRGB          polyLineColor;
TQ3PolyLineData      polyLineData;
TQ3GeometryObject    polyLineObject;

static TQ3Vertex3D   points[4] = {
                       { { -1.0, -0.5, -0.25 }, NULL }, // first vertex
                       { { -0.5,  1.5,  0.45 }, NULL }, // second vertex
                       { {  0.0,  0.0,  0.0  }, NULL }, // third vertex
                       { {  1.5,  1.5,  1.0  }, NULL }  // fourth vertex
                     };

// The polyline has four vertices.
polyLineData.numVertices = 4;
polyLineData.vertices = points;

// Add a color to the line as a whole.
polyLineData.polyLineAttributeSet = Q3AttributeSet_New();
Q3ColorRGB_Set(&polyLineColor, 0.4, 0.2, 0.9);
AttributeSet_AddDiffuseColor(polyLineData.polyLineAttributeSet,
   &polyLineColor);

// Create the polyline.
polyLineObject = Q3PolyLine_New(&polyLineData);

Q3Object_Dispose(polyLineData.polyLineAttributeSet);
```

In Listing 8, we set a color attribute for the entire geometry and for the individual vertices. When you draw the triangle with flat interpolation, the geometry color is used; when you draw it with per-vertex interpolation, however, the vertex attributes take precedence and you can see a color ramp on the triangle (see Figure 8, where the color ramp is approximated in grayscale).

Simple polygon and general polygon objects. Simple polygons and general polygons are planar objects with multiple vertices. Simple polygons must be convex, but general polygons can be either convex or concave. In addition, general polygons can be self-intersecting and have multiple contours.

As was shown in Figure 8, a general polygon can have a "hole" in it, but a simple polygon never does. This is the primary difference between the two geometries. Processing general polygons takes more time than processing simple polygons, so we advise you to use simple polygons whenever possible.

If the geometry you're creating is convex, you should use simple polygons to achieve better performance. If your polygons always have three vertices, however, you should opt for triangles. If you don't know what your geometry looks like (for example, it's being built by the user on the fly and you don't want to check the points), use general polygons and set the complexity flag to kQ3GeneralPolygonShapeHintComplex (see Listing 9). Renderers look at this flag as a hint on how to process the general polygon.

GETTING FANCY

There's nothing wrong with using only simple geometries, as described above. You can build any complex object just with triangles, but from a performance point of

Listing 8. Creating a triangle in a group

```
TQ3ColorRGB          triangleColor;
TQ3GroupObject       model;
TQ3TriangleData      triangleData;
TQ3GeometryObject    triangleObject;

static TQ3Vertex3D   vertices[3] = {{ { -1.0, -0.5, -0.25 }, NULL },
                                    { {  0.0,  0.0,  0.0  }, NULL },
                                    { { -0.5,  1.5,  0.45 }, NULL }};

triangleData.vertices[0] = vertices[0];
triangleData.vertices[1] = vertices[1];
triangleData.vertices[2] = vertices[2];
triangleData.triangleAttributeSet = Q3AttributeSet_New();
Q3ColorRGB_Set(&triangleColor, 0.8, 0.5, 0.2);
AttributeSet_AddDiffuseColor(triangleData.triangleAttributeSet,
   &triangleColor);

triangleData.vertices[0].attributeSet = Q3AttributeSet_New();
triangleData.vertices[1].attributeSet = Q3AttributeSet_New();
triangleData.vertices[2].attributeSet = Q3AttributeSet_New();
Q3ColorRGB_Set(&triangleColor, 1.0, 0.0, 0.0);
AttributeSet_AddDiffuseColor(triangleData.vertices[0].attributeSet,
   &triangleColor);

Q3ColorRGB_Set(&triangleColor, 0.0, 1.0, 0.0);
AttributeSet_AddDiffuseColor(triangleData.vertices[1].attributeSet,
   &triangleColor);

Q3ColorRGB_Set(&triangleColor, 0.0, 0.0, 1.0);
AttributeSet_AddDiffuseColor(triangleData.vertices[2].attributeSet,
   &triangleColor);

// Create the triangle and group.
triangleObject = Q3Triangle_New(&triangleData);
model = Q3OrderedDisplayGroup_New();
if (triangleObject != NULL) {
   Q3Group_AddObject(model, triangleObject);
   Q3Object_Dispose(triangleObject);
}

Q3Object_Dispose(triangleData.vertices[0].attributeSet);
Q3Object_Dispose(triangleData.vertices[1].attributeSet);
Q3Object_Dispose(triangleData.vertices[2].attributeSet);
Q3Object_Dispose(triangleData.triangleAttributeSet);
```

view that's not always the best thing to do. When your object is made up of faces that share vertices, it's a good idea to use a representation that allows the graphics system to reuse the vertex information (such as lighting calculations) for the shared vertices.

With a box, for example, each vertex is shared by three faces, where each face is made up of two triangles. If we draw the box as a bunch of triangles, QuickDraw 3D would have to perform the same lighting calculations on each vertex up to six times. If, on

Listing 9. Creating polygons

```
TQ3PolygonData                polygonData;
TQ3GeneralPolygonData         genPolyData;
TQ3GeometryObject             polygonObject, generalPolygonObject;
TQ3GeneralPolygonContourData  contours[2];
TQ3ColorRGB                   color;

static TQ3Vertex3D  polyVertices[4] = {
                       { { -1.0,  1.0, 0.0 }, NULL },
                       { { -1.0, -1.0, 0.0 }, NULL },
                       { {  1.0, -1.0, 0.0 }, NULL },
                       { {  1.0,  1.0, 0.0 }, NULL }
                    },
                    genPolyHoleVertices[4] = {
                       { { -0.5,  0.5, 0.0 }, NULL },
                       { { -0.5, -0.5, 0.0 }, NULL },
                       { {  0.5, -0.5, 0.0 }, NULL },
                       { {  0.5,  0.5, 0.0 }, NULL }
                    };

polygonData.numVertices = 4; polygonData.vertices = polyVertices;
polygonData.polygonAttributeSet = NULL;
polygonObject = Q3Polygon_New(&polygonData);

contours[0].numVertices = 4; contours[0].vertices = polyVertices;
contours[1].numVertices = 4; contours[1].vertices = genPolyHoleVertices;
genPolyData.numContours = 2; genPolyData.contours = contours;
genPolyData.shapeHint = kQ3GeneralPolygonShapeHintComplex;
genPolyData.generalPolygonAttributeSet = Q3AttributeSet_New();
Q3ColorRGB_Set(&color, 1.0, 1.0, 1.0);
AttributeSet_AddDiffuseColor(genPolyData.generalPolygonAttributeSet,
   &color);

polyVertices[1].attributeSet = Q3AttributeSet_New();
polyVertices[2].attributeSet = Q3AttributeSet_New();
Q3ColorRGB_Set(&color, 0.0, 0.0, 1.0);
AttributeSet_AddDiffuseColor(polyVertices[1].attributeSet, &color);
Q3ColorRGB_Set(&color, 0.0, 1.0, 1.0);
AttributeSet_AddDiffuseColor(polyVertices[2].attributeSet, &color);

genPolyHoleVertices[0].attributeSet = Q3AttributeSet_New();
genPolyHoleVertices[2].attributeSet = Q3AttributeSet_New();
Q3ColorRGB_Set(&color, 1.0, 0.0, 1.0);
AttributeSet_AddDiffuseColor(genPolyHoleVertices[0].attributeSet, &color);
Q3ColorRGB_Set(&color, 1.0, 1.0, 0.0);
AttributeSet_AddDiffuseColor(genPolyHoleVertices[2].attributeSet, &color);

generalPolygonObject = Q3GeneralPolygon_New(&genPolyData);
Q3Object_Dispose(genPolyData.generalPolygonAttributeSet);
Q3Object_Dispose(polyVertices[1].attributeSet);
Q3Object_Dispose(polyVertices[2].attributeSet);
Q3Object_Dispose(genPolyHoleVertices[0].attributeSet);
Q3Object_Dispose(genPolyHoleVertices[2].attributeSet);
```

the other hand, we represent the box as a box primitive or mesh object, the lighting calculations are performed only once per vertex. (However, if you attach vertex colors or face attributes, such as normals or colors, the calculations need to be performed more often.)

Here we show how to use two composite geometries — trigrid and mesh objects — as well as UV parameterization, which you may need to supply if you want to apply a texture to a trigrid or mesh.

Trigrid objects. Trigrids are a collection of triangles that share vertices. We create a trigid in Listing 10.

Listing 10. Creating a trigrid

```
TQ3ColorRGB          triGridColor;
TQ3GroupObject       model;
TQ3TriGridData       triGridData;
TQ3GeometryObject    triGridObject;
unsigned long        numFacets, i;

static TQ3Vertex3D   vertices[12] = {{ { -1.0, -1.0,  0.0 }, NULL },
                                    ... // 10 more lines of vertex data
                                    { {  0.7,  1.0,  0.5 }, NULL }};

triGridData.numRows = 3; triGridData.numColumns = 4;
triGridData.vertices = vertices;
triGridData.triGridAttributeSet = Q3AttributeSet_New();
Q3ColorRGB_Set(&triGridColor, 0.8, 0.7, 0.3);
AttributeSet_AddDiffuseColor(triGridData.triGridAttributeSet,
   &triGridColor);

numFacets = (triGridData.numRows - 1) * (triGridData.numColumns - 1)
   * 2;
triGridData.facetAttributeSet =
   malloc(numFacets * sizeof(TQ3AttributeSet));
for (i = 0; i < numFacets; i++) {
   triGridData.facetAttributeSet[i] = NULL;
}
Q3ColorRGB_Set(&triGridColor, 1.0, 0.0, 0.5);
triGridData.facetAttributeSet[5] = Q3AttributeSet_New();
AttributeSet_AddDiffuseColor(triGridData.facetAttributeSet[5],
   &triGridColor);

triGridObject = Q3TriGrid_New(&triGridData);
```

UV parameterization. Texturing allows you to have more realistic looking models. For texturing to work, the geometry must have *UV parameters* on its vertices, which may have to be supplied by you. The UV parameters are two floating-point values (U and V) that correlate a location on the geometry to a point in the picture of the texture (see Figure 9).

The convention for QuickDraw 3D is to start the UV parameters at 0.0,0.0 at the bottom left, with U increasing toward the right and V increasing upward. You supply the UV parameterization as a collection of vertex attributes.

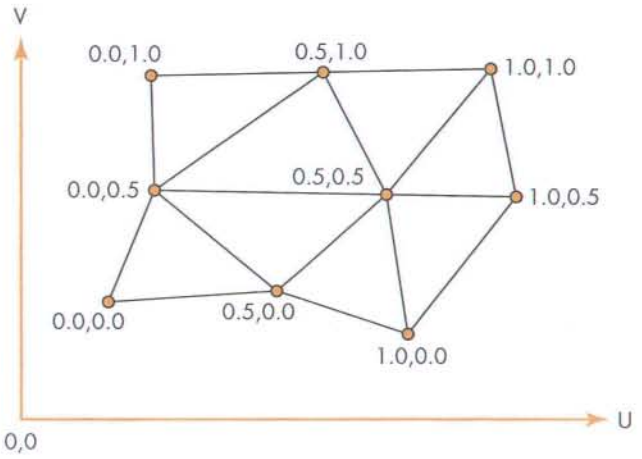


Figure 9. UV parameters on a trigrid's vertices for texture mapping

Once a UV parameterization has been applied to a surface's vertices, the surface can be texture mapped. There are several steps to texturing surfaces with QuickDraw 3D. In general, you'll already have a texture stored in a pixel map somewhere. What you need to do is create a texture shader (of type TQ3TextureObject) and add it into your display group before you add the geometry you want to shade.

Listing 11 is a general-purpose routine for adding a texture shader to a group. It's interesting for a number of reasons: it shows how to search a group for particular objects (in this case, an existing shader that it will replace), how to edit items within a group, and how to add new items.

Listing 11. Routine to texture-map an object

```
TQ3Status AddTextureToGroup(TQ3GroupObject theGroup, TQ3StoragePixmap *textureImage)
{
   TQ3TextureObject  textureObject;
   TQ3GroupPosition  position;
   TQ3Object         firstObject;

   // Create a texture object.
   textureObject = Q3PixmapTexture_New(textureImage);
   if (textureObject) {
      if (Q3Object_IsType(theGroup, kQ3GroupTypeDisplay) == kQ3True) {
         // If the group is a display group...
         Q3Group_GetFirstPosition(theGroup, &position);
         Q3Group_GetPositionObject(theGroup, position, &firstObject);
         if (Q3Object_IsType(firstObject, kQ3SurfaceShaderTypeTexture) == kQ3True) {
            TQ3TextureObject  oldTextureObject;
            TQ3StoragePixmap  oldTextureImage;
            // Replace existing texture by new one.
            Q3TextureShader_GetTexture(firstObject, &oldTextureObject);
            Q3PixmapTexture_GetPixmap(oldTextureObject, &oldTextureImage);
            Q3Object_Dispose(oldTextureObject);
            Q3TextureShader_SetTexture(firstObject, textureObject);
            Q3Object_Dispose(textureObject);
```

(continued on next page)

Listing 11. Routine to texture-map an object *(continued)*

```
        } else {
           TQ3ShaderObject   textureShader;
           // Create texture shader and add it to group.
           textureShader = Q3TextureShader_New(textureObject);
           if (textureShader) {
              Q3Object_Dispose(textureObject);
              Q3Group_AddObjectBefore(theGroup, position, textureShader);
              Q3Object_Dispose(textureShader);
           } else
              return (kQ3Failure);
        }
        Q3Object_Dispose(firstObject);
     } else if (Q3Object_IsType(theGroup, kQ3DisplayGroupTypeOrdered) == kQ3True) {
        // If the group is an ordered display group...
        TQ3ShaderObject   textureShader;
        Q3Group_GetFirstPositionOfType(theGroup, kQ3ShapeTypeShader, &position);
        if (position) {
           Q3Group_GetPositionObject(theGroup, position, &firstObject);
           if (Q3Object_IsType(firstObject, kQ3SurfaceShaderTypeTexture) == kQ3True) {
              TQ3TextureObject  oldTextureObject;
              TQ3StoragePixmap  oldTextureImage;
              // Replace existing texture by new one.
              Q3TextureShader_GetTexture(firstObject, &oldTextureObject);
              Q3PixmapTexture_GetPixmap(oldTextureObject, &oldTextureImage);
              Q3Object_Dispose(oldTextureObject);
              Q3TextureShader_SetTexture(firstObject, textureObject);
              Q3Object_Dispose(textureObject);
           } else {
              // Create texture shader and add it to group.
              textureShader = Q3TextureShader_New(textureObject);
              if (textureShader) {
                 Q3Object_Dispose(textureObject);
                 Q3Group_SetPositionObject(theGroup, position, textureShader);
                 Q3Object_Dispose(textureShader);
              } else
                 return (kQ3Failure);
           }
        } else {
           // Create texture shader and add it to group.
           textureShader = Q3TextureShader_New(textureObject);
           if (textureShader) {
              Q3Object_Dispose(textureObject);
              Q3Group_AddObject(theGroup, textureShader);
              Q3Object_Dispose(textureShader);
           } else
              return (kQ3Failure);
        }
     }
  return (kQ3Success);
  } else         // If pixmap shader not created...
     return (kQ3Failure);
}
```

Mesh objects. Listing 12 shows the key components needed to create a simple mesh geometry. We create a mesh consisting of two faces, with one of them having a hole. We also add UV parameters to the vertices so that we can texture-map the mesh. Figure 10 shows the texture map and the resulting textured mesh.

Listing 12. Creating a mesh

```
TQ3GroupObject BuildMesh(void)
{
   TQ3ColorRGB          meshColor;
   TQ3GroupObject       model;
   TQ3Vertex3D          vertices[9] = {
      { { -0.5,  0.5,  0.0 }, NULL },    { { -0.5, -0.5,  0.0 }, NULL },
      { {  0.0, -0.5,  0.3 }, NULL },    { {  0.5, -0.5,  0.0 }, NULL },
      { {  0.5,  0.5,  0.0 }, NULL },    { {  0.0,  0.5,  0.3 }, NULL },
      { { -0.4,  0.2,  0.0 }, NULL },    { {  0.0,  0.0,  0.0 }, NULL }
   };
   TQ3Param2D           verticesUV[9] = {
      { 0.0, 1.0 }, { 0.0, 0.0 }, { 0.5, 0.0 },
      { 1.0, 0.0 }, { 1.0, 1.0 }, { 0.5, 1.0 },
      { 0.1, 0.8 }, { 0.5, 0.5 }, { 0.1, 0.4 }
   };
   TQ3MeshVertex        meshVertices[9];
   TQ3GeometryObject    meshObject;
   TQ3MeshFace          meshFace;
   TQ3AttributeSet      faceAttributes;
   unsigned long        i;

   meshObject = Q3Mesh_New();
   Q3Mesh_DelayUpdates(meshObject);
   for (i = 0; i < 9; i++) {
      TQ3AttributeSet   vertexASet;
      meshVertices[i] = Q3Mesh_VertexNew(meshObject, &vertices[i]);
      vertexASet = Q3AttributeSet_New();
      AttributeSet_AddSurfaceUV(vertexASet, &verticesUV[i]);
      Q3Mesh_SetVertexAttributeSet(meshObject, meshVertices[i],
         vertexASet);
      Q3Object_Dispose(vertexASet);
   }
   faceAttributes = Q3AttributeSet_New();
   Q3ColorRGB_Set(&meshColor, 0.3, 0.9, 0.5);
   AttributeSet_AddDiffuseColor(faceAttributes, &meshColor);
   meshFace = Q3Mesh_FaceNew(meshObject, 6, meshVertices,
      faceAttributes);
   Q3Mesh_FaceToContour(meshObject, meshFace, Q3Mesh_FaceNew(meshObject,
      3, &meshVertices[6], NULL));
   Q3Mesh_ResumeUpdates(meshObject);
   model = Q3OrderedDisplayGroup_New();
   Q3Group_AddObject(model, meshObject);
   Q3Object_Dispose(faceAttributes);
   Q3Object_Dispose(meshObject);
   return (model);
}
```

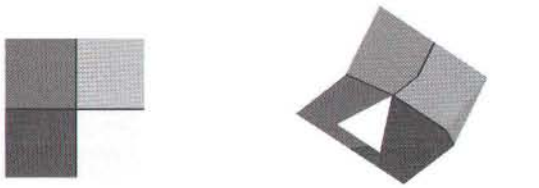


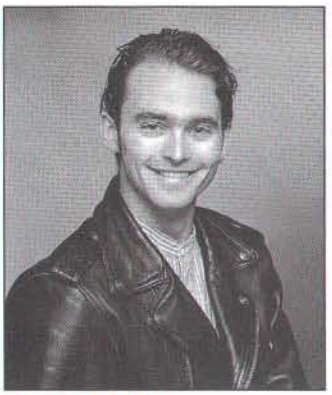
Figure 10. Texture map applied to a mesh

Q3Mesh_DelayUpdates and Q3Mesh_ResumeUpdates, used in Listing 12, are two very important routines. Mesh objects can often contain hundreds and even thousands of vertices. When you're building a complex model, we advise that you turn off updates to the internal ordering of the mesh data, so that building the mesh takes as little time as possible. The difference between doing this and not doing this can be, in the case of a complex model containing 3000 polygons, several minutes when Q3Mesh_DelayUpdates is not called, compared with 3 seconds when it is called (on a mid-level computer).

WHAT DO YOU WANT TO BUILD TODAY?

We hope that the hints in this article will save you some time and help you in your development process. We've been pleasantly surprised by some of the applications in which developers have been putting QuickDraw 3D to use; for example, a European developer used QuickDraw 3D to produce 3D representations of his code profiler application's data. Learning the basics of QuickDraw 3D's geometries is the first step toward mining the rich seam of functionality that QuickDraw 3D offers.

Thanks to our technical reviewers Tom Dowdy, Tim Monroe, and Philip Schneider. ♦



DAVE EVANS

BALANCE OF POWER

Power Macintosh: The Next Generation

The Power Macintosh computer just keeps moving forward. The latest generation brings greatly improved performance and adds the PCI expansion bus and the PowerPC 603 and 604 processors. Software changes that improve performance include the following:

- an improved 680x0 emulator
- a native Resource Manager
- native networking (Open Transport)
- native device drivers
- an improved Memory Manager

I'll describe these new features and discuss how you can maintain compatibility with the new Power Macintosh computers and with future changes to the Mac OS.

THE IMPROVED EMULATOR

First delivered with the Power Macintosh 9500 computer, the new emulator improves on the original in one key way: it actually recompiles 680x0 code into native PowerPC code. Since large portions of the Mac OS are still in 680x0 code, this new emulator speeds up most common operations and offers significant improvements for 680x0 code with tight loops.

Recompiling doesn't mean converting 680x0 instructions one for one into PowerPC instructions. Fully emulating a 680x0 instruction still takes a few PowerPC instructions. But recompiled code is more efficient and optimized. The original emulator had to decipher each instruction every time it was executed, but recompiled code from the new emulator is analyzed once and then executed many times.

Because it takes extra time to recompile code, the emulator doesn't immediately translate all 680x0 code. It operates just like its predecessor until it encounters a loop or similar repetition. Then, instead of emulating the same code repeatedly, it translates the instructions into native code and caches the result. Subsequent calls to that code simply execute the native translation, greatly improving performance.

The cache of translated 680x0 code must stay coherent with memory, much like the caches on the Motorola 68040 processor. Therefore, whenever your software modifies code or changes application jump tables, you should flush the instruction cache. (See the Macintosh Technical Note "Cache as Cache Can" (HW 6) for a more detailed description of cases where flushing the instruction cache is necessary.) In the past you could call Gestalt and check the processor type to flush only on a 68040. Since the new emulator supports only the 68020 instruction set — and Gestalt will indicate that a 68020 is installed — you should now flush any time you modify code or change jump tables.

The best way to flush 680x0 code in the cache is with FlushCodeCacheRange, which flushes only the invalid portion of the emulator's cache. FlushInstructionCache also works but can degrade performance by wastefully purging recompiled code that's still valid. These routines are documented in *Inside Macintosh: Memory*. The C prototype for FlushCodeCacheRange is as follows:

```
OSErr FlushCodeCacheRange(void *address,
   unsigned long count);
```

In 680x0 assembly, you would use

```
MyFlushCodeCacheRange Proc
   ; On entry A0 = address, D0 = # of bytes
   ; Trashes A0, A1, D0. Result in D0, Z bit set.
   ;
   movea.l      D0,A1      ; # bytes in A1
   moveq        #$9,D0     ; selector
   _HWPriv                 ; A098
   tst.w        D0         ; result == noErr
   rts
```

OTHER SOFTWARE CHANGES

The first Power Macintosh computer ported critical portions of the Macintosh Toolbox to native PowerPC

DAVE EVANS and fellow Apple engineer Rus Maxham rode 2000 miles on their motorcycles this summer. They journeyed through the lush Central Valley of California, the blistering heat of the southern Arizona deserts, and the neon glitz of Las Vegas. Along the way they enjoyed the camaraderie of fellow bikers and were rescued in their hour of need by a sympathetic motorcycling couple who housed them as Rus rebuilt his BMW's rear drive assembly.•

code. Ultimately we'll take all of the Mac OS native, but for now we've focused on areas that most increase overall performance. So, starting with the Power Macintosh 9500, we've added a native Resource Manager, the native Open Transport networking stack, and native device drivers. I'll discuss each of these in turn and then mention improvements to the Modern Memory Manager.

Even though many calls to the Resource Manager are bound by I/O bottlenecks, porting the Resource Manager to native PowerPC code still substantially improves performance. Often to complete a request the Resource Manager need only look up existing information and return it, and even if file I/O is required the data is often in the system disk cache. For these reasons, many Resource Manager calls will execute much faster on the new machines.

Native Open Transport networking provides a stream-based interface for networking that's independent of the network protocol. You can now implement a variety of network solutions without concerning yourself with protocol details. Documentation on Open Transport is provided on this issue's CD.

Native device drivers provide both a performance improvement and an improved system programming interface (SPI). This SPI is available with all PCI-based Macintosh computers, starting with the Power Macintosh 9500. For more information on these drivers, see the article "Creating PCI Device Drivers" in *develop* Issue 22 and *Designing PCI Cards and Drivers for Power Macintosh Computers*, available from APDA.

Although not new, the native Modern Memory Manager has been improved in two important ways:

- Many of the routines are now implemented as "fat" binaries instead of all native code. When your 680x0 code calls the Memory Manager, it will now execute 680x0-based routines, eliminating the Mixed Mode environment switch once needed to call the native routines. Reducing the number of these switches can measurably improve performance.
- The bus error handlers have been removed, significantly increasing the performance of many of the simple Memory Manager calls and allowing a number of the calls to be made into fat traps. Bugs discovered during the process of removing the handlers have been fixed.

Handles passed to the Memory Manager now go through a rigorous check before they can affect other Memory Manager data structures; however, without the nearly foolproof bus error handling, it's a little more likely that you'll pass an invalid address and crash. If you crash in the MemoryMgr code fragment while testing on the new Power Macintosh computers, you probably passed an invalid pointer or handle. You can use the Debugging Modern Memory Manager to aggressively catch these application errors.

Note also that the bus error handlers would allow system (and even application) heaps to become corrupted, deteriorating the overall user experience without causing the machine to crash. This is much less likely to happen now, but if structures do get corrupted other than by the Memory Manager, a system crash will result.

Also available starting with the latest Power Macintosh machines is support for very large hard disk volumes. In the past, only 2-gigabyte volumes were allowed; then with System 7.5 we relaxed that restriction to 4-gigabyte volumes. But many of you were still hungry for more, so now we allow up to 2 terabytes (that's 2000 gigabytes) of file system address space per volume. Unless you're developing utilities and drivers compatible with the new volume sizes, though, you really don't need to pay attention to the new large-volume support, because the API remains unchanged. The only time an application might want to take advantage of the new support is when it wants to know before attempting to save to disk whether there's enough free space on the volume. Even in this case, the application won't be able to save a file bigger than the existing limit of 2 GB, and the old version of GetVInfo will return values that are "high-water marked" at 2 GB for compatibility reasons, even if more space is available.

If you really do want to know how much space is available, you can do so through an extension to the File Manager API. We extended the API because the existing 32-bit size information was too small to address volumes and files larger than 4 GB. You'll use the following new routine to get 64-bit sizes:

```
pascal OSErr PBXGetVolInfo(XVolumeParam
                paramBlock, Boolean async);
```

This routine takes an extended VolumeParam structure, named XVolumeParam, which you'll find declared in an updated Files.h interface file on the CD. Before using this routine, be sure to call Gestalt with the gestaltFSAttr selector; if the response parameter has the gestaltFSSupports2TBVolumes bit set, the new routine is available. Note that there are also extended Read and Write calls for drivers that want to support volumes larger than 4 GB.

PCI AND NUBUS

Starting with the PCI-based Power Macintosh computers, support for the NuBus™-specific Slot Manager goes away. Some applications used to call the Slot Manager directly to get video and other device information. This will no longer work, so we've provided better methods: the Display Manager API has been extended for all the video device information you'll need, and the new Name Registry API will give you device information independent of the specific expansion bus implementation.

One example of the improved Display Manager API is the way you get display modes for video devices. With the Slot Manager this took a lot of code, but the Display Manager gives you one encompassing routine:

```
pascal OSErr DMNewDisplayModeList(
                GDHandle theGDevice,
                unsigned long reserved,
                unsigned long *modeCount,
                DMListType *theDisplayModeList,
                unsigned long modeListFlags);
```

With this and other new Display Manager routines, you can avoid the Slot Manager altogether when gathering display information. But if you must access other device information, you can use the bus-neutral Name Registry, which manages a tree of device objects that you can access as a linked list. Look for the new header files (Displays.h and NameRegistry.h) on this issue's CD.

MAINTAINING COMPATIBILITY

As Apple improves the Mac OS, compatibility with the documented APIs and SPIs is ensured — but don't assume that if your application runs fine on existing machines, it will continue to do so in the future. We can't ensure complete compatibility if application code makes invalid assumptions or uses unsupported parts of the Mac OS. There are some things you can do to help ensure that your applications will run on future versions of the Mac OS.

First, use only the officially documented APIs. For example, don't assume that the Z status bit will be set correctly on exit from a trap unless it's documented. As we take more traps native, the 680x0 status register becomes irrelevant and such checks break. Here's an example of 680x0 code that now breaks because it assumes the Z status bit will be set by Get1Resource:

```
move.l    #'DAVE',-(sp)
clr.w     -(sp)
_Get1Resource
beq.s     error        ; BAD!
```

You also shouldn't expect results in registers if the trap isn't documented to return them there. It's true that some traps used to accidentally exit with useful data in register D0 or A0, but if that's not documented as part of the API it won't be supported in the future.

Second, test your software using EvenBetterBusError, the Debugging Modern Memory Manager, and any other debugging tools that are appropriate (look in the Testing & Debugging folder on the CD). Stress-testing your software with these tools will catch many errors that otherwise would go unnoticed. EvenBetterBusError catches most stray references to nil, such as writing to location 0 or using nil pointers and handles. The Debugging Modern Memory Manager catches those occasions when you damage a heap or pass invalid addresses.

Finally, as I've said in previous columns, don't use RS/6000 POWER instructions in your native code. Although the PowerPC 601 processor supports many of them, the new 603 and 604 processors do not. We've made an attempt to emulate the POWER instructions in software for these new processors, but this emulation is very expensive. When a 603 or 604 encounters one of these now-illegal instructions, it stops everything and calls our new illegal-instruction handler, which recognizes the instruction that was used and attempts to use a valid one instead. This operation is very time consuming; if your performance-critical code includes POWER instructions, you'll see a severe slowdown. As described in this column in *develop* Issue 21, you should use the DumpXCOFF tool to check your code for any POWER instructions.

NEW DIRECTIONS

Apple will continue to take advantage of RISC technology and will both improve existing performance and add new functionality. Make sure your code uses documented interfaces so that it will stay compatible and run on future generations of the Power Macintosh. And be sure to check out Open Transport and PCI device drivers — they're exciting new directions that will take you closer to the next generation of the Mac OS today.

Thanks to Bill Knott, Eric Traut, and Jack Valois for reviewing this column.•

Special thanks to Randy and Peggy Marlatt of Flagstaff, Arizona, for road support.•

Implementing Shared Internet Preferences With Internet Config

Having to enter the same Internet preferences, such as e-mail address and news server, into multiple applications is bothersome not just for users, but also for developers who must create the user interface associated with them. The Internet Configuration System (IC) provides a simple user application for setting preferences, and an API for getting the preferences from a database that's shared by all applications. It's easy to add IC support to your application and take advantage of the flexibility gained by IC's use of the Component Manager — a valuable technique in itself.

QUINN "THE ESKIMO!"

Preferences, like nuclear weapons, proliferate. At times it seems that the major developers are engaged in a "preferences race," where each one tries to gain the upper hand by adding a dozen new preferences in each new release. Like the arms race, the preferences race is obviously counterproductive, even dangerous, and yet no one knows how to stop it.

Some of the worst offenders are Internet-related applications. How many times have you had to enter your e-mail address into a configuration window? And what about your preferred type and creator for JPEG files? Doesn't this just seem like a waste of your time? The Internet Configuration System, or Internet Config for short, spares everyone this trouble. And it spares developers the complexities of implementing these preferences in each application.

This article takes you inside Internet Config. Take a good look at the design: IC implements its shared library as a component, and uses switch glue to provide a default implementation if the component is absent. Using the Component Manager to implement shared libraries is a helpful technique not just for IC, but for other APIs as well. Note too that Internet Config is useful for more than its name implies. For example, the extension-to-file-type mapping database is useful for any program that deals with "foreign" file systems. Indeed, IC is a perfectly valid mechanism for storing private preferences that have nothing to do with the Internet.

Although IC is intended as an abstract API, all its source code is placed in the public domain — a condition of its development. This lets me illustrate the text with

QUINN "THE ESKIMO!" (quinn@cs.uwa.edu.au) has a first name but, when asked about it, his usual response is "I could tell you but then I'd have to kill you!" He programs for a living with the Department of Computer Science at the University of Western Australia, but on weekends he gets together with Peter N. Lewis and programs for *fun*. The Internet Configuration System is a product of these misspent recreational hours. Quinn writes in Pascal using a Dvorak keyboard on a Macintosh Duo that he carries around on his bicycle, and he's still trying to figure out how to use this minority status to his economic advantage. •

snippets from the actual implementation and gives you full access to the source code. Both the IC user's kit and the IC developer's kit, which contain code and documentation, are included on this issue's CD. Note that Internet Config was developed independently and is not supported by Apple.

The latest versions of the kits are always available from the ftp sites ftp://ftp.share.com/pub/internet-configuration/ and ftp://redback.cs.uwa.edu.au/Others/Quinn/Config/. In addition, the user kit is available from UMich and Info-Mac mirrors around the world.•

As with any new piece of software intended to be widely adopted, Internet Config needs developer support in order to be successful. I hope this article raises the awareness of IC in the developer community and prompts some of you to support it.

INTERNET CONFIG FROM THE OUTSIDE

Before going inside Internet Config, it's important to know how the system works as a whole. The best way to do this is to get a copy of the Internet Config application and run it (there's a copy on this issue's CD), but if you're too relaxed to do that right now, keep reading for a description of the basics. We'll look at IC first from the user's perspective and then from the programmer's point of view.

THE USER'S PERSPECTIVE

To the user, Internet Config is a proper Macintosh application. It supports the standard menu commands New, Open, Save, Save As, and so on. The only difference is that the files it operates on are preferences files. Figure 1 shows Internet Config and its related files.

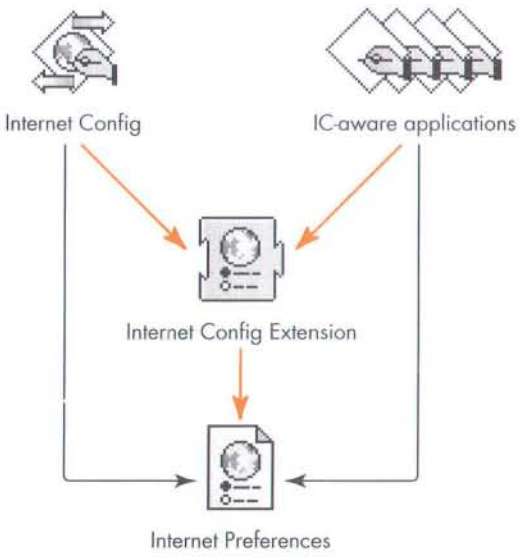


Figure 1. Internet Config and its related files — what the user sees

The first time the Internet Config application is run, it installs the Internet Config Extension into the Extensions folder and creates a new, blank Internet Preferences file in the Preferences folder. It then displays the main window, shown in Figure 2, which allows the user to edit the preferences.

Each of the buttons in the main window displays another window containing a group of related preferences. For example, the Personal button brings up the window shown

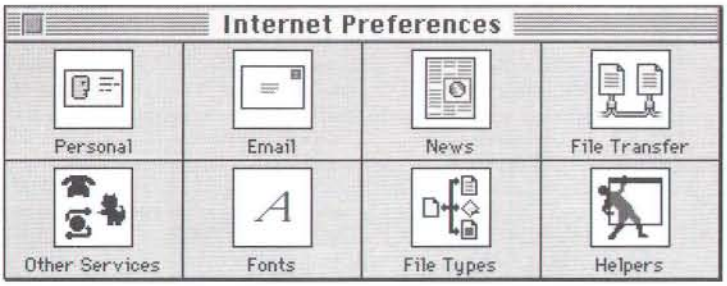


Figure 2. The Internet Config application's main window

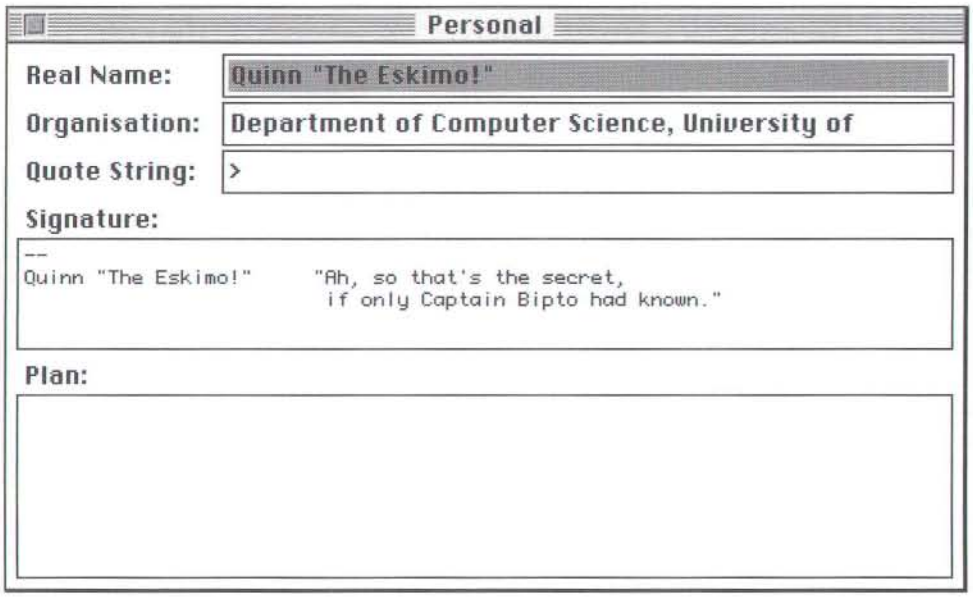


Figure 3. The Personal preferences window

in Figure 3. The user enters preferences into each of these windows and then quits and saves the preferences.

From this point on, the user never has to enter those preferences again. Any IC-aware program the user runs simply accesses the preferred settings without requiring them to be reentered. This makes the user very happy (we presume).

Users can even run IC-aware applications "out of the box" — they don't have to run Internet Config first. If the Internet Config Extension isn't installed, IC-aware client applications access the Internet Preferences file directly instead of through the extension (as shown by the black arrows in Figure 1). The way this is done is described later in the section "The Inner Workings of an API Routine."

THE PROGRAMMER'S PERSPECTIVE

To programmers, Internet Config consists of a set of interface files that define the API, and a library to be statically linked to their programs. IC can be used from all of the common Macintosh development environments: MPW, THINK, and Metrowerks; Pascal and C; and 680x0 and PowerPC. The examples in this article, like IC itself, were written in THINK Pascal.

What's in an IC preference. Before getting to the details of the API, you need to know more about IC preferences. In IC, a *preference* is an item of information that's useful to the client application program. Each preference has three components: its key, its data, and its attributes.

- The *key* is a Str255 that identifies the preference. You can use the key to fetch the data and attributes.
- The *data* is an untyped sequence of bytes that represents the value of the preference. The data's structure is determined by the client program. The structures of the common preferences are defined in the IC programming documentation.
- The *attributes* represent information about the preference that's supplementary to the preference data, such as whether the preference is read/write or read-only.

In the e-mail address preference, for example, the key is the string "Email". If you pass this string into IC, it returns the preference's data and attributes. By convention, the data for the key "Email" is interpreted as a Pascal string containing the user's preferred e-mail address.

IC's core API routines. Internet Config has the following core API routines. Although the API has a lot more depth, these four routines are all you need to program with IC.

```
FUNCTION ICStart (VAR inst: ICInstance; creator: OSType): ICError;

FUNCTION ICStop (inst: ICInstance): ICError;

FUNCTION ICFindConfigFile (inst: ICInstance; count: Integer;
                           folders: ICDirSpecArrayPtr): ICError;

FUNCTION ICGetPref (inst: ICInstance; key: Str255; VAR attr: ICAttr;
                    buf: Ptr; VAR size: LongInt): ICError;
```

The ICStart routine is always called first. Here you pass in your application's creator code so that future versions of IC can support application-dependent preferences. ICStart returns a value of type ICInstance; this is an opaque type that must be passed to every other API call. ICStop is called at the termination of your application to dispose of the ICInstance you obtained with ICStart.

ICFindConfigFile is called immediately after ICStart. IC uses this routine to support applications with double-clickable user configuration files, a common phenomenon among Internet applications. If you need to support these files, see the IC programming documentation; otherwise, just pass in 0 for the count parameter and nil for the folders parameter.

The ICGetPref routine takes a preference key and returns the preference's attributes in **attr** and its data in the buffer pointed to by **buf**. The maximum size of the buffer is passed in as **size**, which is adjusted to the actual number of bytes of preference data.

The simplest example. The program in Listing 1 demonstrates the simplest possible use of IC technology. All it does is write the user's e-mail address to the standard output. This program calls the four core API routines: it begins by calling ICStart and terminates with an ICStop call; it calls ICFindConfigFile with the default parameters and uses ICGetPref to fetch the value of a specific preference — in this case the user's e-mail address.

Listing 1. The simplest IC-aware program

```
PROGRAM ICEmailAddress;
   { The simplest IC-aware program. It simply outputs the user's }
   { preferred e-mail address. }

   USES
      ICTypes, ICAPI, ICKeys;    { standard IC interfaces }

   VAR
      instance:   ICInstance; { opaque reference to IC session }
      str:        Str255;     { buffer to read e-mail address into }
      str_size:   LongInt;    { size of above buffer }
      junk:       ICError;    { place to throw away error results }
      junk_attr:  ICAttr;     { place to throw away attributes }
BEGIN
   { Start IC. }
   IF ICStart(instance, '????') = noErr THEN BEGIN
      { Specify a database, in this case the default one. }
      IF ICFindConfigFile(instance, 0, NIL) = noErr THEN BEGIN
         { Read the real name preferences. }
         str_size := sizeof(str);   { 256 bytes -- a similar construct }
                                    { wouldn't work in C }
         IF ICGetPref(instance, kICEmail, junk_attr, @str, str_size)
                  = noErr THEN BEGIN
            writeln(str);
         END;  { IF }
      END;  { IF }
      { Shut down IC. }
      junk := ICStop(instance);
   END;  { IF }
END.  { ICEmailAddress }
```

INSIDE INTERNET CONFIG

The IC API just described is really all you need to know to make your program IC-aware; now we'll get into the guts of Internet Config to see how it achieves its magic. We'll look first at its underlying design and then at how its internal structures work together.

THE IC DESIGN: A SIMPLE, EXPANDABLE SYSTEM

The design requirements for Internet Config evolved during early discussions of what an Internet configuration system might look like (see "How Internet Config Came to Be"). These requirements guided the development process and form the basic structure of Internet Config — an efficient, expandable system that's easy to use and easy to support.

Internet Config can accept sweeping changes while maintaining API compatibility, and it allows for patches to support future extensions and bug fixes. We couldn't achieve such expandability with a simple shared preferences implementation, and the consequent loss of simplicity caused a lot of debate during the development process.

The need for simplicity was implicit from the beginning. To add support for Internet Config, application developers have to revise their code. Developers tend to be lazy

HOW INTERNET CONFIG CAME TO BE

Designing Internet Config was a complicated business. The process began in March 1994 with a discussion on the Usenet newsgroup comp.sys.mac.comm. Many people thought simplifying Internet configuration was a good idea, but few agreed how best to achieve the goal, or indeed what the goal was.

We set up a mailing list to swap ideas, and discussion continued apace for weeks. One of the biggest issues was the disparity between the problems we wanted to solve and the ones we *could* solve given our limited resources.

After a week or two of thrashing out the requirements, Peter N. Lewis, Marcus Jager, and I proposed the first API. A few weeks later we shipped the first implementation of the Internet Config Extension.

The problem IC solves is actually quite simple, so it didn't take long to implement the design. As usual, however, it took some time to go from a working implementation to a final product — we shipped Internet Config 1.0 in December 1994. Though we've made minor additions and changes, the initial design survives to this day.

— hey, I mean that as a compliment — and generally prefer simple systems to complicated ones. Developer support is critical for success, so we kept the system simple. Still, it isn't so simple as to compromise the need for expandability.

As we've already seen, IC has several other interesting design features. The API supports applications with double-clickable user configuration files. The Internet Config user application accesses all the Internet preferences through the API, and is thereby isolated from the implementation details. IC-aware applications work even if the Internet Config Extension isn't installed. We even included support for System 6 (much as we resented it).

IC'S INTERNAL STRUCTURES

As you can see in Figure 4, the Internet Config application and IC-aware client programs have very similar internal structures. In fact, except for a few artifacts

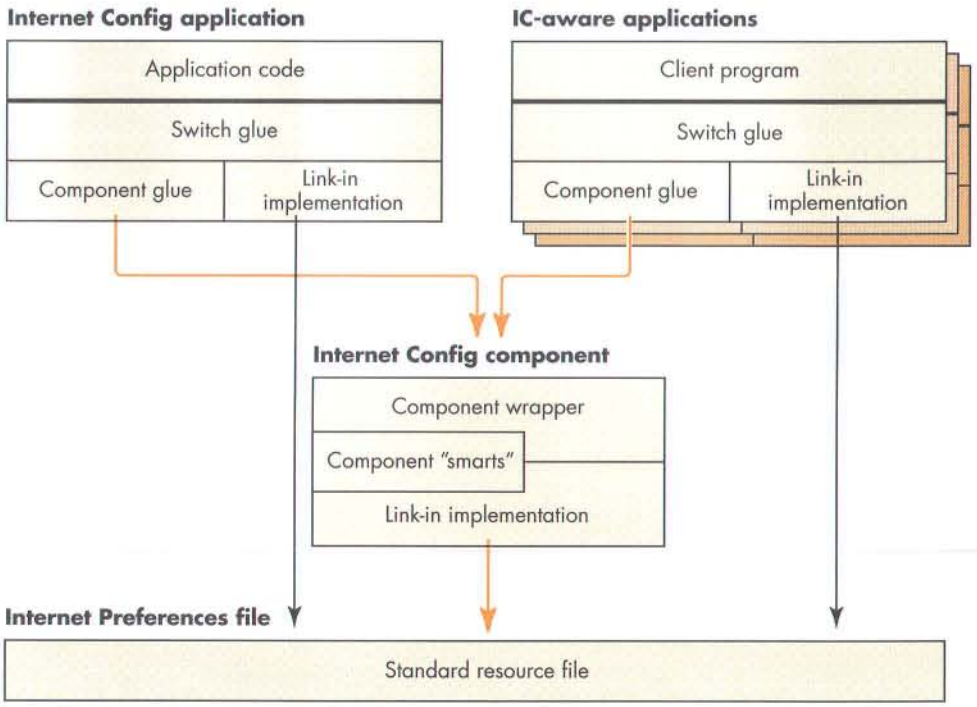


Figure 4. Inside the Internet Config entities — what the programmer sees

caused by implementing "safe saving," the Internet Config application uses the standard API to modify the Internet Preferences file. The Internet Config component, which the user sees as the Internet Config Extension, is basically a shared library of routines implemented as a component (see "The IC Component and Shared Libraries on the Macintosh").

The switch glue is a common interface that applications use to call IC. This glue decides whether the Internet Config component is available and, if it is, routes all calls through to it. If the component isn't present, the calls are routed through to the link-in implementation, which then does the work.

This switching mechanism satisfies two design requirements. It allows the API to be patched by replacing or overriding the Internet Config component. It also allows IC-aware programs to work even if the component isn't installed; they simply fall back to using the link-in implementation.

THE INNER WORKINGS OF AN API ROUTINE

Now we'll look more closely at how the Start and GetPref routines are implemented in each part of the Internet Config system. We'll trace these two calls from the top level, where they're called by the client program, all the way down to the link-in implementation, where the real action takes place.

This section is quite technical; if you're not interested in the implementation details, you might want to just skim through it. Many of the details are provided for illustrative purposes only. Take heed! *If you write client programs that rely on these details, they will break in future revisions of IC.* The public interface to IC is defined in the IC programming documentation.

We'll start with the switch glue and proceed through the standard call path. On the way we'll examine the component glue, wrapper, and "smarts," and finally, the link-in implementation. The path is convoluted but rewards you with both data and code abstraction.

Start and GetPref appear in each part of the system, and each appearance has a specific purpose, as we'll see in a moment. To keep things straight, various instances

THE IC COMPONENT AND SHARED LIBRARIES ON THE MACINTOSH

The Internet Config component is essentially a shared library of routines. So why implement it as a component? The answer lies in the confused state of shared libraries on the Macintosh.

When we started writing IC we knew we'd need a shared library. The problem was not that the system didn't have a shared library mechanism, but that it had too many. At the time there were four Apple shared library solutions, each with its unique drawbacks: the Component Manager wasn't a "real" shared library system; the Apple Shared Library Manager (ASLM) had limited availability and lacked PowerPC support and developer tools; the Code Fragment Manager (CFM) lacked 680x0 support; and the System Object Model (SOM) lacked any availability.

These days life is a little better. ASLM now works on the PowerPC platform, CFM is being ported to the 680x0 platform, SOM is imminent, and Apple has issued a clear statement of direction on shared libraries, centered on CFM.

But statements of direction don't solve problems — they just clear up confusion. The shared library problem persists. When I was writing this article someone asked me for advice about which shared library mechanism to use. My recommendation today is the same as at the start of the IC project: use the Component Manager. It's still the only solution that has the developer tools, has 680x0 and PowerPC support, and is already installed on most users' machines.

of the same routine are prefixed to denote which part of the system they're in. The prefixes are listed in Table 1, which shows the various specifications for the GetPref routine as an example. (Note that these specifications vary only in the name's prefix and the type of the first parameter. The "R" in the ICR prefix indicates that these routines actually use the Resource Manager to modify the preferences; all the other routines are glue.)

Table 1. Routine name prefixes

Prefix	Part of System	First Parameter	GetPref Specification
IC	Standard API (switch glue)	ICInstance	FUNCTION ICGetPref (inst: ICInstance; key: Str255; VAR attr: ICAttr; buf: Ptr; VAR size: LongInt): ICError;
ICC	Component API (component glue)	ComponentInstance	FUNCTION ICCGetPref (inst: ComponentInstance; key: Str255; VAR attr: ICAttr; buf: Ptr; VAR size: LongInt): ICError;
ICCI	Component internal	globalsHandle	FUNCTION ICCIGetPref (inst: globalsHandle; key: Str255; VAR attr: ICAttr; buf: Ptr; VAR size: LongInt): ICError;
ICR	Link-in implementation	VAR ICRRecord	FUNCTION ICRGetPref (var inst: ICRRecord; key: Str255; VAR attr: ICAttr; buf: Ptr; VAR size: LongInt): ICError;

THE SWITCH GLUE

The switch glue relies on ICRRecord, the central data structure of IC, shown in Listing 2. The first field of ICRRecord, **instance**, is a ComponentInstance, which normally holds the connection to the Internet Config component. If the component is installed, the instance field holds the connection to it; the rest of the fields are ignored because the component has a separate ICRRecord in its global variables. If the component isn't installed, the instance field is nil, and the link-in implementation uses the rest of the fields to hold the necessary state (as we'll see later).

Listing 2. ICRRecord

```
TYPE
   ICRRecord = RECORD
      { This entire record is completely private to the }
      { implementation!!! Your code will break if you depend }
      { on the details here. You have been warned. }
      instance: ComponentInstance;
               { nil if no component available; if not nil, }
               { then rest of record is junk }
      ...   { other fields to be discussed later }
      END;
   ICRRecordPtr = ^ICRRecord;
```

The switch glue for the application's Start routine, ICStart, is shown in Listing 3. The first thing ICStart does is attempt to allocate an ICRRecord; if it succeeds, it then tries to open a connection to the component with the component glue routine ICCStart. ICCStart either succeeds, setting the internal instance field to the connection to the component, or fails and returns an error. If ICCStart returns an error, ICStart falls back to using the link-in implementation by calling ICRStart. If ICRStart fails, Internet Config fails to start up; ICStart sets **inst** to nil and returns an error.

Listing 3. The switch glue for Start

```
FUNCTION ICStart (VAR inst: ICInstance; creator: OSType): ICError;
   VAR
      err:  ICError;
BEGIN
   inst := NewPtr(sizeof(ICRRecord));
   err := MemError;
   IF err = noErr THEN BEGIN
      err := ICCStart(ICRRecordPtr(inst)^.instance, creator);
      IF err <> noErr THEN BEGIN
         err := ICRStart(ICRRecordPtr(inst)^, creator);
      END;  { IF }
      IF err <> noErr THEN BEGIN
         DisposePtr(inst);
         inst := NIL;
      END;  { IF }
   END;  { IF }
   ICStart := err;
END;  { ICStart }
```

Listing 4. The switch glue for GetPref

```
FUNCTION ICGetPref (inst: ICInstance; key: Str255; VAR attr: ICAttr;
                        buf: Ptr; VAR size: LongInt): ICError;
BEGIN
   IF ICRRecordPtr(inst)^.instance <> NIL THEN BEGIN
      ICGetPref := ICCGetPref(ICRRecordPtr(inst)^.instance,
                                key, attr, buf, size);
   END
   ELSE BEGIN
      ICGetPref := ICRGetPref(ICRRecordPtr(inst)^, key, attr, buf, size);
   END;  { IF }
END;  { ICGetPref }
```

The switch glue for GetPref, and all the other API routines for that matter, is very simple. All it does is consult the internal instance field to determine whether ICStart successfully connected to the component. If so, it calls through to the component glue routine ICCGetPref; otherwise, it calls through to the link-in implementation routine ICRGetPref. This is shown in Listing 4.

The switch glue implementations of both Start and GetPref do a lot of casting between ICInstance and ICRRecordPtr, because the ICRRecordPtr type describes details of the implementation that shouldn't "leak out" to the client's view of IC. The client programs know only of ICInstance, which is an opaque type. The explicit casts could have been avoided with some preprocessor tricks, but we decided to include them longhand for clarity.

THE COMPONENT GLUE

The component glue calls the Internet Config component. In the component glue for the Start routine, shown in Listing 5, Internet Config attempts to connect to the IC component by calling the Component Manager routine OpenDefaultComponent.

Listing 5. The component glue for Start

```
FUNCTION ICCStartComponent (inst: ComponentInstance; creator: OSType):
                           ICError;
INLINE                { standard Component Manager glue }
   $2F3C, $04, $0,    { move.l    #$0004_0000,-(sp) }
   $7000,             { moveq.l   #0,d0 }
   $A82A;             { _ComponentDispatch }

FUNCTION ICCStart (VAR inst: ComponentInstance; creator: OSType):
                           ICError;
   VAR
      err, junk:  ICError;
      response:   LongInt;
BEGIN
   inst := NIL;
   IF Gestalt(gestaltComponentMgr, response) = noErr THEN BEGIN
      inst := OpenDefaultComponent(internetConfigurationComponentType,
                                   internetConfigurationComponentSubType);
   END;  { IF }
   IF inst = NIL THEN BEGIN
      err := badComponentInstance;
   END
   ELSE BEGIN
      err := ICCStartComponent(inst, creator);
      IF err <> noErr THEN BEGIN
         junk := CloseComponent(inst);
         inst := NIL;
      END;  { IF }
   END;  { IF }
   ICCStart := err;
END;  { ICCStart }
```

If the Internet Config component isn't installed or can't be opened for any other reason, the routine sets **inst** to nil and fails with a badComponentInstance error. Remember that the calling code, ICStart, will notice this error code and fall back to the link-in implementation, as shown in Listing 4.

If the routine successfully opens a connection to the Internet Config component, it calls the ICCStartComponent routine, which is standard Component Manager glue that calls the component's initialization routine.

The component glue version of GetPref is a lot simpler. It's just a standard piece of Component Manager glue, as shown in Listing 6. The inline instructions of the component glue for GetPref translate into the piece of assembly code shown in Listing 7.

You can read more about the Component Manager and its dispatch mechanism in *Inside Macintosh: More Macintosh Toolbox*.

Calling components from PowerPC code is not described in this article or in *Inside Macintosh: More Macintosh Toolbox*. You can find out how to do this by reading the Macintosh Technical Note "Component Manager Version 3.0" (QT 5).•

Listing 6. The component glue for GetPref

```
FUNCTION ICCGetPref (inst: ComponentInstance; key: Str255;
                        VAR attr: ICAttr; buf: Ptr;
                        VAR size: LongInt): ICError;
INLINE              { standard Component Manager glue }
   $2F3C, $10, $6,  { move.l    #$0010_0006,-(sp) }
   $7000,           { moveq.l   #0,d0 }
   $A82A;           { _ComponentDispatch }
```

Listing 7. Disassembling the component glue

```
move.l   #$0010_0006,-(sp)    ; push the routine selector (6) and the
                              ; number of bytes of parameters (16)
moveq.l  #0,d0                ; _ComponentDispatch routine selector to
                              ; call a component function
_ComponentDispatch            ; call the component through the Component
                              ; Manager
```

THE COMPONENT WRAPPER

Now let's look inside the Internet Config component at the component wrapper (Listing 8). The component wrapper's basic function is to dispatch all of the IC component's routines based on the selector in **params.what**; it uses a big CASE statement to determine the routine's address and then calls the routine with the Component Manager function CallComponentFunctionWithStorage. The Component Manager is smart enough to sort out the parameters at this stage.

Most of the API routines are immediately dispatched by the component wrapper to an internal routine that simply calls the link-in implementation to do the work. For example, the ICCIGetPref routine, shown in Listing 9, calls through to ICRGetPref, changing only the first parameter.

Listing 8. Sections of IC's component wrapper

```
FUNCTION Main (VAR params: ComponentParameters; storage: Handle):
                        ComponentResult;
{ Inside Macintosh has params as a value parameter when it should be }
{ a VAR parameter. Don't make this mistake. }
   VAR
      proc: ProcPtr;
      s:    SignedByte;
BEGIN
   proc := NIL;
   CASE params.what OF
      { Dispatch the routines required by the Component Manager. }
      ...   { routines omitted for brevity }
      { Dispatch the routines that make up the IC API. }
      kICCStart:
         proc := @ICCIStart;
```

(continued on next page)

Listing 8. Sections of IC's component wrapper *(continued)*

```
      kICCGetPref:
         proc := @ICCIGetPref;
      ...   { remaining IC API routines omitted for brevity }
      OTHERWISE
         Main := badComponentSelector;
   END;  { case }
   IF proc <> NIL THEN BEGIN
      IF storage <> NIL THEN BEGIN
         s := HGetState(storage);
         HLock(storage);
      END;  { IF }
      Main := CallComponentFunctionWithStorage(storage, params, proc);
      IF (storage <> NIL) AND
            (params.what <> kComponentCloseSelect) THEN BEGIN
         HSetState(storage, s);
      END;  { IF }
   END;  { IF }
END;  { Main }
```

Listing 9. The component wrapper for GetPref

```
FUNCTION ICCIGetPref (globals: globalsHandle; key: Str255; VAR attr:
                           ICAttr; buf: Ptr; VAR size: LongInt): ICError;
BEGIN
   ICCIGetPref := ICRGetPref(globals^^.inst, key, attr, buf, size);
END;  { ICCIGetPref }
```

So you can see that there are two ways to call ICRGetPref, either from the component's internal routine ICCIGetPref or from the switch glue's ICGetPref. This is consistent with the design outlined in Figure 4. Of course, these routines call two different copies of the code, one linked into the program and one linked into the component.

THE COMPONENT "SMARTS"

The component "smarts" are wedged between the component wrapper and the link-in implementation. Most component wrapper routines don't have smarts; they call straight through to the link-in implementation. Adding smarts to a routine allows it to work better than its link-in cousin without the need to maintain two versions of the routine.

A good example of a smart routine is the component wrapper version of the Start routine, ICCIStart (Listing 10). This fixes a potential localization problem associated with the link-in implementation with a clever sleight of hand. ICCIStart is basically the same as ICCIGetPref in that it immediately calls through to its link-in implementation equivalent. But then it does something tricky: the component calls itself to get the default filename for the Internet Preferences file. For the gory details of why this is "smart," see "Smart Components for Smart People."

One thing to note is that when ICCIStart calls the component to get the default filename, it doesn't do so directly, but instead uses the component glue to call its current_target global variable. Targeting is cool Component Manager technology

Listing 10. A smart component wrapper

```
FUNCTION ICCIStart (globals: globalsHandle; creator: OSType): ICError;
{ Handle the start request, which is basically a replacement for the }
{ open because we need another parameter, the calling application's }
{ creator code. }
   VAR
      err:  OSErr;
BEGIN
   err := ICRStart(globals^^.inst, creator);
   IF err = noErr THEN BEGIN
      err := ICCDefaultFileName(globals^^.current_target,
                        globals^^.inst.default_filename);
   END;  { IF }
   ICCIStart := err;
END;  { ICCIStart }
```

that allows you to write override components (more on this later in "Override Components").

With each new version of Internet Config, the component implementation gets smarter than the link-in implementation. Component smarts are used in IC 1.0 to improve ease of localization; in IC 1.1, they're also used to improve targetability. In a future version of IC, component smarts may be used to implement a preference cache.

THE LINK-IN IMPLEMENTATION

It may be hard to imagine, but everything you've seen so far is glue. The code that does the real work in IC is the link-in implementation. The link-in implementation sees a different view of the ICRRecord, one that contains enough fields to store all the data that the implementation requires. This extended view of the ICRRecord is shown in Listing 11.

The instance field is still there but the link-in implementation ignores it. It's the subsequent fields that are of interest. Most of them are easy to understand with the help of their comments.

SMART COMPONENTS FOR SMART PEOPLE

Because Internet Config needs to know the default filename of the Internet Preferences file when it creates a new preferences file, and because all filenames should be stored in resources so that they can be localized, the default filename should be stored in a resource. This approach is fine for the component, which can get at its resource file with OpenComponentResFile, but doesn't work for the link-in implementation since it can be linked in to a variety of applications.

We considered working around this by requiring all applications to add a resource specifying the name, but this would force all of our developers to add resources to their applications, and the resource ID might clash with their existing resources. The biggest disadvantage, however, is that IC clients are not necessarily applications and may not even have resource files associated with them.

So we solved this problem by making the component version of IC smarter than the link-in version. The link-in version sets default_filename to "Internet Preferences" and leaves it at that, while the component version calls itself to get the correct filename from the resource file.

Listing 11. The full ICRRecord in the link-in implementation

```
TYPE
   ICRRecord = RECORD
      { This entire record is completely private to the }
      { implementation!!! Your code will break if you depend }
      { on the details here. You have been warned. }
      instance: ComponentInstance;
               { nil if no component available; if not nil, then rest }
               {  of record is junk }
      have_config_file: Boolean;
               { determines whether any file specification calls, that }
               { is, ICFindConfigFile or ICSpecifyConfigFile, have been }
               { made yet; determines whether the next field is valid }
      config_file: FSSpec;
               { our chosen database file }
      config_refnum: Integer;
               { a place to store the resource refnum }
      perm: ICPerm;
               { the permissions the user opened the file with }
      inside_begin: Boolean;
               { determines if config_refnum is valid }
      default_filename: Str63;
               { the default IC filename }
      END;
   ICRRecordPtr = ^ICRRecord;
```

The link-in implementation for the Start routine initializes the remaining ICRRecord fields, as shown in Listing 12.

Listing 12. The link-in implementation for Start

```
FUNCTION ICRStart (VAR inst: ICRRecord; creator: OSType): ICError;
   VAR
      junk: ICError;
BEGIN
   inst.have_config_file := false;
   inst.config_file.vRefNum := 0;
   inst.config_file.parID := 0;
   inst.config_file.name := '';
   inst.config_refnum := 0;
   inst.perm := icNoPerm;
   junk := ICRDefaultFileName(inst, inst.default_filename);
   ICRStart := noErr;
END;  { ICRStart }

FUNCTION ICRDefaultFileName (VAR inst: ICRRecord; VAR name: Str63):
                              ICError;
BEGIN
   name := ICdefault_file_name;
   ICRDefaultFileName := noErr;
END;  { ICRDefaultFileName }
```

Finally, there's the link-in implementation for GetPref, portions of which are shown in Listing 13. The actual implementation is a bit long, so the listing leaves out a lot of messing around with resources, bytes, pointers, attributes, and so on. The basic operation of the routine is simple, however: it checks its parameters, opens the preferences file (by calling ICRForceInside), gets the preference, closes the preferences file, and returns.

Listing 13. The link-in implementation for GetPref

```
FUNCTION ICRGetPref (VAR inst: ICRRecord; key: Str255; VAR attr: ICAttr;
                        buf: Ptr; VAR size: LongInt): ICError;
   VAR
      err, err2:              ICError;
      max_size, true_size:    LongInt;
      old_refnum:             Integer;
      prefh:                  Handle;
      force_info:             Boolean;
BEGIN
   max_size := size;
   size := 0;
   attr := ICattr_no_change;
   prefh := NIL;
   err := ICRForceInside(inst, icReadOnlyPerm, force_info);
   IF (err = noErr) AND (inst.config_refnum = 0) THEN BEGIN
      err := icPrefNotFoundErr;
   END;  { IF }
   IF (err = noErr) AND ((key = '') OR
         ((max_size < 0) AND (buf <> nil))) THEN BEGIN
      err := paramErr;
   END;  { IF }
   IF err = noErr THEN BEGIN
      old_refnum := CurResFile;
      UseResFile(inst.config_refnum);
      err := ResError;
      IF err = noErr THEN BEGIN
         ...   { lots of resource hacking here }
         UseResFile(old_refnum);
      END;  { IF }
   END;  { IF }
   IF prefh <> NIL THEN BEGIN
      ReleaseResource(prefh);
   END;  { IF }
   err2 := ICRReleaseInside(inst, force_info);
   IF err = noErr THEN BEGIN
      err := err2;
   END;  { IF }
   ICRGetPref := err;
END;  { ICRGetPref }
```

TOWARD THE FUTURE

The future . . . where Macintosh applications glide along the information superhighway, seamlessly perceiving the user's every preference. You'd better hope your applications are IC aware!

Internet Config is a very flexible system that can expand in several dimensions. Indeed, some are already being explored — in particular, the use of components to maintain and extend the system. And we're looking forward to seeing IC extended in ways we never anticipated.

OVERRIDE COMPONENTS

One of the coolest features of the Component Manager is targeting — one component can capture another and override it. This effectively prevents external programs from using the captured component, while still allowing it to be called by the override component. Very much like inheritance in object-oriented design, this technology lets you write a very simple component that captures the Internet Config component so that you can patch just one routine. For example, the Internet Config RandomSignature extension overrides the ICGetPref routine. If an IC client requests the signature preference, the extension randomly chooses one from a collection of signatures.

The possibilities for override components are endless. Let's say your organization wants to preconfigure all news clients to access a central news server. You can do this by writing a simple override component that watches for programs getting the NNTPHost preference and returns a fixed read-only preference value. This way, all IC-aware news readers use the correct host but can't change it. As we say in the system software business, it's a wonderful third-party developer opportunity.

TOTAL BODY SWAP

Because all client programs call Internet Config through a well-defined API, it's possible to write a replacement for IC and gain complete control of the system. Imagine that you're tired of having the same preferences in all your IC-aware applications. You can change them by writing a replacement that conforms to the existing API. First, replace the Internet Config component with a smarter one that's capable of storing a set of preferences for each application and returning the right preferences to the right application. Then replace the Internet Config application with a much more sophisticated application that can manage multiple sets of preferences, and your job is done. All IC-aware programs will automatically benefit without recompilation.

Or suppose you want to store your user preferences on a central server and access them through some network protocol. Again, IC lets you do it. You could replace the Internet Config component with a network-aware one, and establish the user's identity in some way, perhaps by requiring the user to log on before using any IC-aware programs. You could then choose to use either a Macintosh application to administer the server or tools from the server's native environment.

STAYING CURRENT

No program is ever finished, nor is any program ever 100% bug free. Internet Config is getting better all the time, and you can update to the newest, improved version with a minimum of fuss. When the application detects that its version of the Internet Config Extension is out of date, it simply installs the new one. Because all IC-aware programs are dynamically linked to the component contained within this extension, they automatically receive the update without having to be recompiled.

By the time you read this article, IC 1.1 should be released and busily updating old versions of the Internet Config Extension around the globe. IC 1.1 offers many improvements and bug fixes, including an extended API and a shell for writing override components easily. Share and enjoy!

RECOMMENDED READING

If you want to find out more about Internet Config itself, the following documents may be of interest:

- "Using the Internet Configuration System" by Quinn, *MacTech Magazine*, April 1995.
- *Internet Configuration System: User Documentation* and *Internet Configuration System: Programming Documentation* by Quinn, in the IC User's Kit and IC Developer's Kit, respectively (1994). These kits are provided on this issue's CD.
- "Internet Config FAQ" by Quinn (1994–1995). Available from the ftp site ftp://redback.cs.uwa.edu.au/Others/Quinn/Config/IC_FAQ.txt.

Here's where you can find out more about components, the technology Internet Config is based on:

- *Inside Macintosh: More Macintosh Toolbox* (Addison-Wesley, 1993).
- Macintosh Technical Note "Component Manager Version 3.0" (QT 5).
- "Be Our Guest: Components and C++ Classes Compared" by David Van Brink, *develop* Issue 12.
- "Inside QuickTime and Component-Based Managers" by Bill Guschwan, *develop* Issue 13.
- "Somewhere in QuickTime: Derived Media Handlers" by John Wang, *develop* Issue 14.
- "Managing Component Registration" by Gary Woodcock, *develop* Issue 15.

Finally, if you're interested in the mindset of Internet Config's authors, you can do no better than to read the following:

- *He Died With a Felafel in His Hand* by John Birmingham (The Yellow Press, 1994).
- *The UNIX-HATERS Handbook* by Simson Garfinkel, Daniel Weise, and Steven Strassmann (IDG Books, 1994).
- http://www.cm.cf.ac.uk/Movies/

Thanks to our technical reviewers Peter Hoddie, Peter N. Lewis, Jim Reekes, and Greg Robbins. Internet Config is a joint development by Peter N. Lewis and Quinn, with design input from Marcus Jager. We'd like to thank all of those on the Internet Config mailing list and the developers who are supporting the system.•

The Internet Config mailing list is dedicated to discussing the technical details of Internet Config. You can subscribe by sending mail to listserv@list.peter.com.au with the body of the message containing "subscribe config Your Real Name."•

MPW TIPS AND TRICKS

Customizing Source Control With SourceServer

TIM MARONEY

When two engineers on a team edit the same source file at the same time, the resulting chaos can be terrible to behold. Source control was invented to mitigate the problem. Most Macintosh programmers are familiar with the MPW Shell's Check In and Check Out dialogs, and with its Projector commands. The next frontier of custom source control involves SourceServer, a nearly faceless application that implements most of the Projector commands. MPW scripts are easy to write, but they're no match for the power, speed, and friendliness of compiled software. SourceServer exports Projector commands as Apple events, allowing source control from compiled software without launching the MPW Shell in all its pomp and splendor.

Popular third-party development environments often send Apple events to SourceServer for integrated source control. You can also use SourceServer to customize Projector beyond what you might have thought possible. For instance, you can drag source control, kicking and screaming, into the modern world of user experience with drop-on applications. In this column, I'll show you how to check a file in or out with a simple drag and drop, and how to use SourceServer for other things as well. The sample code is provided on this issue's CD; SourceServer is distributed, with documentation, on the MPW Pro and E.T.O. CDs (available from APDA) and with third-party development systems.

APPLE EVENTS FOR SOURCESERVER

Apple events have many faces, but they're primarily a way of communicating between different applications. Each Apple event encapsulates a message as a command with any number of input parameters; the receiver of the message may return any number of result parameters to the sender. The most basic unit of data is the Apple event *descriptor*, which consists of a type code and a data handle. Apple events are built out of descriptors and are themselves special kinds of complex descriptors.

For an excellent introduction to Apple events, see "Scripting the Finder From Your Application" by Greg Anderson in *develop* Issue 20. •

SourceServer's commands are represented as descriptor lists. Its Apple events are exact duplicates of the MPW Shell's Projector commands, but to avoid the overhead of a full command parser, both the command name and each argument are descriptors in the descriptor list. This saves you the trouble of putting quotes and escapes into arguments that might contain spaces or other special characters. The downside is that you have to expand arguments yourself: you can't pass in MPW wildcard characters, backquoted commands for expansion, or other special constructs.

Creating descriptor lists may sound harder than writing MPW scripts, but that's only because it is. I've provided some utility routines to ease the way, though. Listing 1 shows the utilities and illustrates how to make a command to check out a file for modification. As illustrated in the CheckOut routine in this listing, you call the CreateCommand routine first and then use the AddXArg routines to add arguments.

Some of the utilities take Pascal strings, while others take C strings, which could well be considered bad programming practice. I chose this dubious method not because I'm on drugs, but because Pascal strings and C strings are used in different ways. SourceServer's text descriptors are C strings; when passed to these utilities as string constants, they shouldn't be converted from Pascal format in place, since some compilers put constants in read-only areas. If you're internationally savvy, you may have another objection: string constants themselves are bad practice. However, for better or worse, MPW scripts and tools are not internationalized. Just like aliens in *Star Trek*, all MPW programmers are assumed to speak English.

TIM MARONEY wrote TOPS Terminal and BackDrop, and has been a major contributor to TOPS for Macintosh, FaxPro, and Cachet. He has also contributed to Fiery, the Disney Screen Saver, Ofoto, Colortron, and the Usenet Mac Programmer's Guide. Tim learned computer networking while working on the Andrew and MacIP projects at Carnegie Mellon and studied compiler design in graduate school at Chapel Hill. He has written for all three major operating systems and a few minor ones. On the Macintosh, Tim's code has included applications, INITs, control panels, HyperCard stacks, XCMDs, shared libraries, trap patches, plug-ins, scripts, and things more difficult to characterize. Tim is currently doing contract work at Apple, and is available for parties and special events at a nominal cost. •

Listing 1. Creating SourceServer commands

```
OSErr CreateCommand(AEDesc *command, CString commandText)
/* Begin a new SourceServer command; name of command is in commandText. */
{
   OSErr err = AECreateList(NULL, 0, false, command);
   if (err != noErr) return err;
   err = AddCStringArg(command, commandText);
   if (err != noErr) (void) AEDisposeDesc(command);
   return err;
}

OSErr AddCommentArg(AEDesc *command, StringPtr comment)
/* Add a "-cs comment" argument to a SourceServer command. */
{
   OSErr err;
   if (comment[0] == 0) return noErr;
   err = AddCStringArg(command, "-cs");
   if (err != noErr) return err;
   err = AddPStringArg(command, comment);
   return err;
}

/* Other SourceServer argument utilities */
OSErr AddDirArg(AEDesc *command, short vRefNum, long folderID);
OSErr AddProjectArg(AEDesc *command, StringPtr projectName);
OSErr AddUserArg(AEDesc *command, StringPtr userName);
OSErr AddFullNameArg(AEDesc *command, FSSpec *file);
OSErr AddPStringArg(AEDesc *command, StringPtr string);
OSErr AddCStringArg(AEDesc *command, CString string);

OSErr CheckOut(FSSpec *file, StringPtr userName, StringPtr projectName, StringPtr comment)
/* Create a "Check Out Modifiable" command for SourceServer: */
/* CheckOut -m -cs <comment> -d <dir> -project <project> -u <user> <file> */
{
   OSErr          err;
   AEDesc         command;
   CStringHandle  output = NULL, diagnostic = NULL;

   err = CreateCommand(&command, "CheckOut");
   if (err != noErr) return err;
   err = AddCStringArg(&command, "-m");
   if (err == noErr) err = AddCommentArg(&command, comment);
   if (err == noErr) err = AddDirArg(&command, file->vRefNum, file->parID);
   if (err == noErr) err = AddProjectArg(&command, projectName);
   if (err == noErr) err = AddUserArg(&command, userName);
   if (err == noErr) err = AddPStringArg(&command, file->name);
   if (err == noErr) err = SourceServerCommand(&command, &output, &diagnostic);
   (void) AEDisposeDesc(&command);
   /* Display output or diagnostic text as desired. */
   if (output != NULL) DisposeHandle((Handle) output);
   if (diagnostic != NULL) DisposeHandle((Handle) diagnostic);
   return err;
}
```

While on the subject of programming practice, I must gently reprimand SourceServer for its approach to Apple events, in which script commands are simulated through a single 'cmnd' event. SourceServer's idiosyncratic convention dates from the earliest days of Apple events, and modern guidelines discourage this type of design. An application implementing its own Apple events should designate a different command code for each operation, treating arguments as keyword parameters.

Listing 2 shows how to send an Apple event to SourceServer. It's first necessary to find and perhaps launch the SourceServer application. The snippet called SignatureToApp (by Jens Alfke) on this issue's CD accomplishes this with a single function call. Simply pass in the creator code of SourceServer, which is 'MPSP'.

The event must be created before it can be sent. For SourceServer, there's a single parameter, named keyDirectObject, which is the descriptor list containing the command. After sending the event, you must extract the results. The results of an Apple event are returned as keyword parameters in a reply descriptor. First there's the standard keyErrorNumber parameter, which returns an error code if delivery failed. SourceServer returns three other parameters: The 'stat' parameter contains a second error code; if it's nonzero, SourceServer tried to execute the command and failed. When there's an error, there will be diagnostic output in the 'diag' parameter, a handle containing text from the MPW diagnostic (error) channel. Finally, there's standard output — a handle specified by keyDirectObject — which contains explanatory text.

PROJECTDRAG — DRAG AND DROP SOURCE CONTROL

The Macintosh has always had a drag and drop user experience, but the true power and generality of dragging has been widely recognized only recently. The drag paradigm can even be used for source control. To turn Projector into a drag-savvy system, I've written a set of utilities called ProjectDrag (source code and documentation are provided on this issue's CD). You simply drag and drop icons onto the following miniapplications that make up ProjectDrag, and the corresponding function is performed:

- Check In and Check Out, for checking files in and out
- ModifyReadOnly, for editing a file without checking it out
- Update, for bringing a file or folder up to date, as well as canceling checkouts and modify-read-only changes
- ProjectDrag Setup, for configuring the system

These utilities are based on a drop-on application framework called DropShell (written by Leonard Rosenthol and Stephan Somogyi), also on the CD. When a file is dropped onto an application, the application receives an Open Documents ('odoc') event. DropShell takes care of the rigmarole of receiving this and other required Apple events. The ProjectDrag miniapplications pull the file specifications out of 'odoc' events and create SourceServer commands that operate on the files and folders that were dropped on their icons.

DropShell is also available on the Internet at ftp://ftp.hawaii.edu/pub/mac/info-mac/Development/src/ and at other Info-Mac mirror sites. •

Some setup is required. ProjectDrag needs to know the locations of Projector databases. It maps between project names and Projector database files by keeping aliases to database folders in its Preferences folder. To start using a project, simply drag its ProjectorDB file or the enclosing folder onto ProjectDrag Setup. Projector also needs to know your user name, and your initials or a nickname are used in change comments at the start of files. These are stored in a text file in the Preferences folder. ProjectDrag asks you for this information if it can't find it, or you can launch ProjectDrag Setup and give the Set User Name command.

ProjectDrag is scriptable, unlike SourceServer and the MPW Shell. The miniapplications have an Apple event terminology resource ('aete') to advertise their events to scripting systems. This allows you to add source control commands to any application that lets you add AppleScript scripts to its menus.

ProjectDrag is able to run remotely over a network. This circumvents a limitation of SourceServer, which can only be driven locally. ProjectDrag can receive remote Apple events and then drive a copy of SourceServer that's local to it. Among other uses, this could support an accelerator for Apple Remote Access. Checking a file in or out over ARA takes a few minutes, which is fine, especially for those who find tedium particularly enjoyable. Copying files is faster. With local AppleScript front ends for remote ProjectDrag miniapplications, you could copy files to and from a remote "shadow folder" and initiate SourceServer commands at the remote location, where they would execute over a fast network such as Ethernet.

I like to think that I can solve user interface problems in my sleep. When I was writing ProjectDrag, I had a dream of a better user experience. Instead of miniapplications, ProjectDrag would be a magical system extension that would put a single small icon at

Listing 2. Sending commands to SourceServer

```
OSErr SourceServerCommand(AEDesc *command, CStringHandle *output, CStringHandle *diagnostic)
{
   AppleEvent            aeEvent;
   AERecord              aeReply;
   AEDesc                sourceServerAddress, paramDesc;
   ProcessSerialNumber   sourceServerProcess;
   FSSpec                appSpec;  /* SignatureToApp requires this due to a minor bug */
   long                  theLong, theSize;
   DescType              theType;
   OSErr                 err;

   *output = *diagnostic = NULL;   /* default replies */

   /* Find the SourceServer process and make a descriptor for its process ID. */
   err = SignatureToApp('MPSP', NULL, &sourceServerProcess, &appSpec, NULL,
                        Sig2App_LaunchApplication, launchContinue + launchDontSwitch);
   if (err != noErr) return err;
   err = AECreateDesc(typeProcessSerialNumber, (Ptr) &sourceServerProcess,
                      sizeof(ProcessSerialNumber), &sourceServerAddress);
   if (err != noErr) return err;

   /* Create and send the SourceServer Apple event. */
   err = AECreateAppleEvent('MPSP', 'cmnd', &sourceServerAddress, kAutoGenerateReturnID,
                            kAnyTransactionID, &aeEvent);
   (void) AEDisposeDesc(&sourceServerAddress);     /* done with the address descriptor */
   if (err != noErr) return err;
   err = AEPutParamDesc(&aeEvent, keyDirectObject, command);   /* add the command */
   if (err != noErr) { (void) AEDisposeDesc(&aeEvent); return err; }
   err = AESend(&aeEvent, &aeReply, kAEWaitReply + kAENeverInteract, kAENormalPriority,
                kNoTimeOut, NULL, NULL);
   (void) AEDisposeDesc(&aeEvent);    /* done with the Apple event */
   if (err != noErr) return err;

   /* Check for an error return in the keyErrorNumber parameter. */
   err = AEGetParamPtr(&aeReply, keyErrorNumber, typeInteger, &theType, &theLong,
                       sizeof(long), &theSize);
   if (err == noErr && (err = theLong) == noErr) {
      /* Get the standard output from the keyDirectObject parameter. */
      err = AEGetParamDesc(&aeReply, keyDirectObject, typeChar, &paramDesc);
      if (err == noErr) *output = (CStringHandle) paramDesc.dataHandle;
      /* Get the diagnostic output from the 'diag' parameter. */
      err = AEGetParamDesc (&aeReply, 'diag', typeChar, &paramDesc);
      if (err == noErr) *diagnostic = (CStringHandle) paramDesc.dataHandle;
      /* Get the MPW status from the 'stat' parameter -- it becomes our error return. */
      err = AEGetParamPtr(&aeReply, 'stat', typeInteger, &theType, &theLong,
                          sizeof(long), &theSize);
      if (err == noErr) err = theLong;
   }

   (void) AEDisposeDesc(&aeReply);    /* done with the reply descriptor */
   return err;
}
```

some convenient place on the screen. When you dragged a file onto this icon, it would pop open into a temporary window and show you icons for the various options. Dreams are great for creativity, but it's easier to weigh alternatives when you're awake. After I woke up, I realized that miniapplications will be able to do the same thing.

Here's how: In Copland, the next generation of the Mac OS, the Finder will spring-load folders so that they open automatically when you drag onto them. It will also let you stash commonly used folders at the bottom of the screen, where they appear as short title bars. Drag the ProjectDrag folder to the bottom of the screen and you're set! Since the Finder will be providing my dream interface, there's no point in a lot of trap patching and extensibility infrastructure to accomplish the same thing.

Copland will bring another user experience benefit to ProjectDrag: it's planned that document windows will have a draggable file icon in their title bar, so you'll be able to use ProjectDrag on an open document by dragging the icon from its window.

YOU TAKE IT FROM HERE

You can create programs that use SourceServer for many other tasks. On cross-platform projects, Projector is sometimes used to control both platforms' source folders. This can lead to baroque and error-prone processes. With SourceServer, you can create front ends that do the right thing. They could copy to remote folders over a network, or lock read-only files since the other platform doesn't see Projector's 'ckid' resources.

Quality is an interesting area for source control applications. A quality tool could query Projector databases for the frequency and scope of changes at various stages of the project, correlating them with bug tracking to develop project metrics. Along similar lines, a tool could measure the change rate of various files to assist in what the quality gods refer to as root-cause analysis.

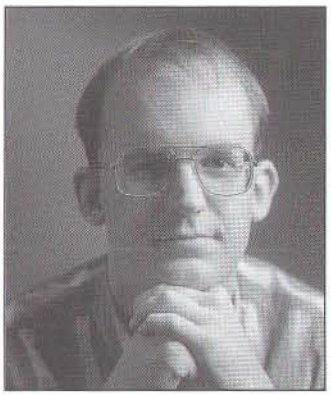
SourceServer is much more than a way for development systems to provide integrated source control. It's great for structuring your internal development process as well!

Thanks to Greg Anderson, Arno Gourdol, and B. Winston Hendrickson for reviewing this column.•

Special thanks to Jens Alfke, Jon Pugh, Leonard Rosenthol, and Stephan Somogyi.•

Multipane Dialogs

As applications grow in power and complexity, so does the tendency to present users with numerous cluttered dialog boxes. To simplify the user interface, developers are moving increasingly to dialogs with multiple panes. This article describes how to implement multipane dialogs that users navigate by clicking in a scrolling list of icons.



NORMAN FRANKE

Dialog boxes with multiple panes ("pages" of controls) are an increasingly popular element of the Macintosh user interface. Like simple dialogs, multipane dialogs can be presented when users need to indicate preferences, set attributes of text or graphic objects, or give specifications for complex operations such as searches or formatting, among other things. By grouping related options and providing a single point of interaction for manipulating them, multipane dialogs simplify life for the user and the developer.

Five different kinds of controls for navigating multipane dialogs are in general use: the scrolling list of icons, the pop-up menu, index tabs (simulating the look of tabs on the tops of file folders in a file cabinet), Next/Previous buttons, and icon button sets. Although there aren't any hard-and-fast rules about when you should use one over another, these considerations (suggested by Elizabeth Moller of Apple's Human Interface Design Center) generally apply:

- Novice users have trouble with pop-up menus, so choose a different kind of control if your target audience includes large numbers of these users.
- Index tabs work well for small numbers of panes, but they may not work well when the tabs start overlapping or the number of panes is variable.
- Next/Previous buttons are a good choice when there's more than one mandatory pane. They make it easy for users to step through mandatory and optional panes in sequence.

The sample application MPDialogs on this issue's CD demonstrates the use of a multipane preferences dialog navigated by clicking in a scrolling list of icons, similar to the Control Panel in System 6 and print dialogs in QuickDraw GX. After describing the user interface presented by this sample program, I'll go into the details of how to implement a similar multipane dialog in your own application. Source code for the routines I'll discuss is also included on the CD. This code requires System 7 and is compatible with both black-and-white and color displays.

NORMAN FRANKE misses the large electrical storms and green things of his native Pennsylvania, but not the humidity. He's using the B.S. in computer science he earned from Carnegie Mellon as he writes Macintosh software for a large national laboratory in northern California. Now working on an M.S. in computer science at Stanford, he enjoys writing sound manipulation software for his Macintosh and watching classic and action/adventure movies in his spare time.•

WHAT THE USER INTERFACE LOOKS LIKE

To experience how multipane dialogs work, run the sample program MPDialogs. When you choose Preferences from the File menu, you'll be presented with the interface shown in Figure 1. This is a good illustration of the elements of a multipane dialog.

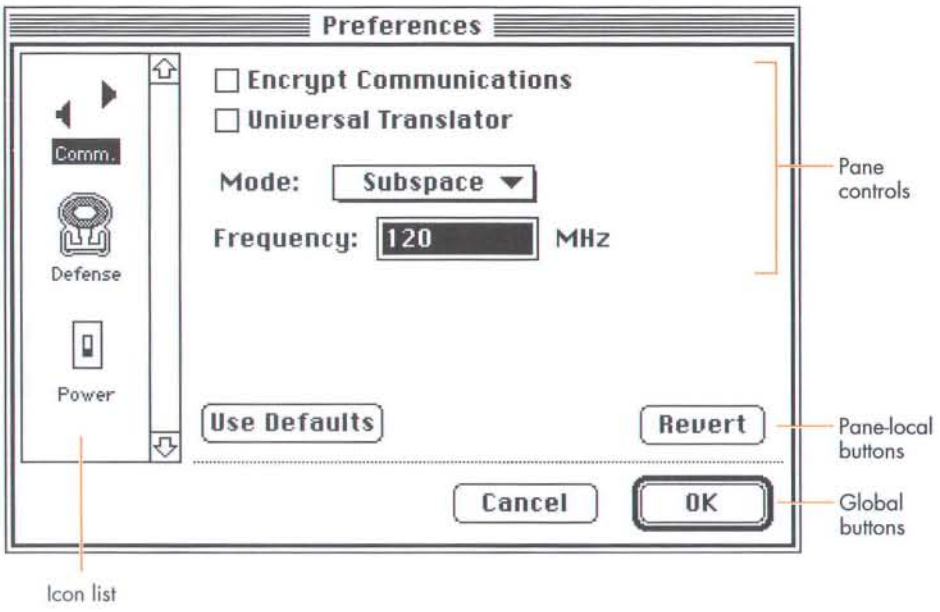


Figure 1. The Communications pane of the sample multipane dialog

The long vertical rectangle on the left side of the dialog box contains the pane selection icon list. Each icon in this scrolling list has a one-word label under it for identification and represents one pane of the dialog, which is displayed when the user clicks the icon. If you click the Defense icon, for instance, you'll see the pane shown in Figure 2. The arrow and tab keys on the keyboard can also be used to change the pane selection; however, if the current pane contains multiple editable text fields, the tab key will work as in a normal dialog and move the cursor to the next text field.

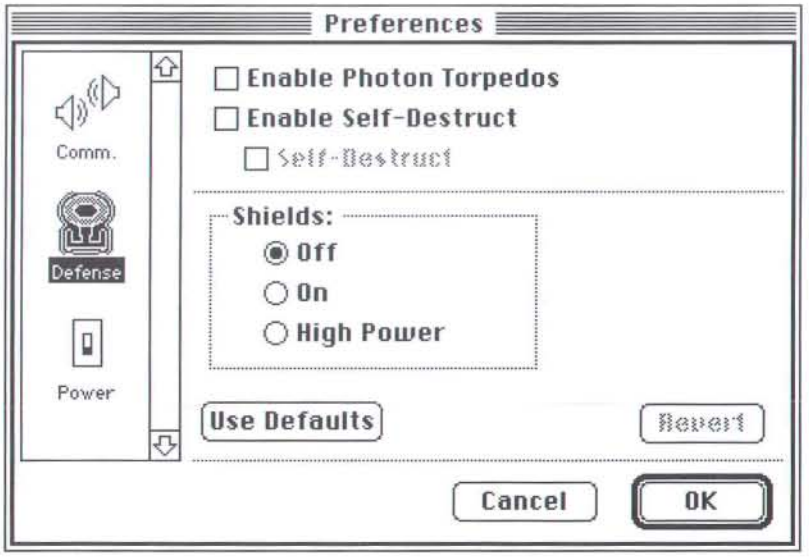


Figure 2. The Defense pane of the sample multipane dialog

The bottom portion of the dialog below the line contains two buttons that act on the dialog as a whole: Cancel and OK. The OK button accepts the settings and Cancel aborts all changes and closes the dialog. The two buttons above the line act only on the current pane and are optional: Revert restores the control values in the current pane to what they were when that pane was last opened, and Use Defaults resets the control values in that pane to factory defaults.

The large region above the buttons is where the pane's controls are placed. The sample code supplied on the CD handles actions for checkboxes, radio button groups, and pop-up menus. Command-key equivalents can be used to toggle checkboxes and radio buttons, in addition to the standard keyboard equivalents for OK (Return/Enter) and Cancel (Escape/Command-period). After experimenting with making changes to the control values in the sample program, you can choose Display from the File menu to see the results of your changes.

A couple of custom capabilities can be added to a pane through optional procedures:

- taking special action such as dimming or undimming other controls when items are clicked
- performing data validation such that if validation fails, the user isn't permitted to change panes or exit the dialog with the OK button

These two capabilities are demonstrated in the sample multipane dialog. When you click the Enable Self-Destruct checkbox in the Defense pane, the Self-Destruct checkbox is undimmed. When you enter nondigits in the editable text field in the Communications pane, data validation fails and you're unable to change panes or click OK.

Note that multipane dialogs, like simple dialogs, can take one of three forms:

- standard modal dialog — a dialog that has a border around it and no title bar, that can't be moved around on the screen, and that stays frontmost as long as it's open
- movable modal dialog — a dialog that has a border around it and a title bar, that can be moved around on the screen, and that stays frontmost as long as it's open and the application is frontmost
- modeless dialog — a dialog that looks and behaves like a normal document window with a title bar and a close box, and that isn't always frontmost

The sample program displays a movable modal dialog, but the code provided supports all three forms.

That's all there is to the interface. For some words of wisdom about things to take into account as you design your own multipane dialogs, see "Tips for Designing Multipane Dialogs." Now we'll move along to the details of how to incorporate the multipane dialog routines on the CD into your own application: the resources you need to define, the calls to make to the main routines to open the dialog and handle events, and the customizing you can do with optional procedures.

DEFINING NEEDED RESOURCES

The first step in incorporating the multipane dialog routines is to define the custom resources the code needs. You'll find ResEdit TMPL templates for all the needed resources on the CD. You can put these in the ResEdit Preferences file to make them available at all times or leave them in the application you're editing.

TIPS FOR DESIGNING MULTIPANE DIALOGS

BY ELIZABETH MOLLER OF APPLE'S HUMAN INTERFACE DESIGN CENTER

A multipane dialog is appropriate only when the panes you're presenting are obviously related to one another in some way. With that caveat in mind, here are some suggestions for making your multipane dialogs easy to understand and use:

- Provide a sentence or title to help clarify your intent. For example, you might precede a scrolling list of icons in a preferences dialog with a sentence like "Select items from this list to set your SurfWriter preferences."
- If you use an icon list, label the icons in your list to help users recognize them.
- Visually separate buttons that apply only to the current pane from those that work on all panes (like OK and Cancel in a modal dialog).
- Don't change the size of the dialog or window as the user navigates from pane to pane. Pick a size that accommodates the pane with the most controls.
- Design the dialog so that changing the settings in one pane doesn't change the settings in another. For example, clicking a checkbox in one pane shouldn't disable a checkbox in another pane, because the user won't see the latter action occur and thus won't understand the cause and effect.
- Be consistent in your use of controls. If you use a particular type of control to mean "choose a setting," for instance, don't use the same type of control to mean "navigate between panes" in the same dialog. Users should be able to easily distinguish controls that navigate through multiple panes from controls that make choices in the dialog.
- Order the panes from mandatory to optional, by frequency of use, from general to specific, or, when no other order is apparent, alphabetically. If there are mandatory fields and controls, be sure to put them in the first pane or step the user through mandatory panes before optional ones.
- When the dialog is closed, remember the pane that was last used, unless there are mandatory controls in a pane. If there is a mandatory pane, it should always be displayed when the dialog reopens.

The first resource that needs to be created is the main DLOG and its associated DITL, which will form the basis for the dialog. A sample is provided in the file MPDialogs Resources that you can simply copy into a new project's resource file. The DITL should include six items, numbered as follows:

1. OK button
2. Cancel button
3. Revert button
4. Use Defaults button
5. a user item that defines the icon list rectangle
6. a hidden static text field for default Command-key equivalents

The Revert and Use Defaults buttons can be moved offscreen to make either of them unavailable. (Alternatively, the buttons can be removed and the control **#defines** in the main header file, MPDialogs.h, can be changed to reflect the new numbering.) The icon list is always displayed vertically, and the rectangle doesn't include the scroll bar. The sample application provides the standard Command-key equivalents for OK and Cancel. The standard equivalents for OK are handled in the code; those for Cancel are handled by means of the hidden static text field, which defines default Command-key equivalents for the rest of the controls in the dialog as well.

A DITL needs to be created for each pane. The first item is a hidden static text field that defines Command-key equivalents for the items in the pane; this is in addition to the default list in the main DITL. See "Code for Dialog Command-Key Equivalents" for details of the syntax.

The items are numbered local to each DITL, so that, for example, the first control would be item 2. All user items in the DITL are set to the DrawGray procedure, which outlines the item's rectangle with either the gray color or a stippled gray pattern, depending on the user's monitor.

Next, a DTL# resource should be created with the same resource ID as the main DLOG resource. It contains a list of the resource IDs of the DITLs that comprise a specific multipane dialog and the text displayed under each icon in the list. Then the icon groups are created; they have the same resource ID as the DITL to which they correspond. Small versions of the icons aren't needed, but color versions should be created for display on color-capable Macintosh computers.

Optional DGRP resources can be created for specifying radio button groups. The resource ID is the same as that of the corresponding pane's DITL. Each DGRP can contain multiple groups per pane, if desired; however, a particular radio button should only be used in a single group. Like the per-pane Command-key equivalent strings, items are numbered local to the DITL.

You should also copy the following:

- the pseudo-CDEF with resource ID 251, which provides support for using the icon list as a control (in the file MPDialogs Resources)
- the LDEF with resource ID 130, which implements the icon list definition for the List Manager (in the file Icon LDEF in the LDEF folder)
- optionally, the 'hdlg' resource and corresponding STR# resource for Balloon Help support (in the file MPDialogs Resources)

You can add Balloon Help to a multipane dialog by adding two help items to the individual DITL resources that make up each pane. One is for the controls in the

CODE FOR DIALOG COMMAND-KEY EQUIVALENTS

The Command-key equivalent code I provide in the sample uses a modified version of KeyEquivFilter, a routine in Utilities.c, which is part of DTS Lib on the CD. It takes these two additional parameters:

- The ID of the static text item that contains the mappings. My dialog code calls this routine twice, once for the bottom buttons and a second time for the items in the pane.
- An offset to add to the item numbers when a hit occurs. This allows the code to use relative item numbering for easier specification of Command-key equivalents in panes.

The static text item is an item-match string that follows the general format *=cxxyyzz* or *ccxxyyzz*. The *=c* matches the character *c*, and *cc* matches the character by its ASCII value. The next number, *xx* (a flag byte with the bits set to specify the modifier keys you're checking for), is logically ANDed with the modifier flags from the key-down event and compared to *yy* (a flag byte with the bits set to specify the values of the modifier keys — for example, you can force the Control key to be up). If this comparison is true and if the character *c* matches the character the user typed, the item *zz* is returned as being hit.

Each item-match string is eight characters long and is separated from other such strings that follow by a comma. The numbers in the strings are hexadecimal and case is significant for character matches.

For example, the hidden static text field that's checked for each pane in the sample application is

```
=.190102,1B190102,1B190002
```

The first item-match string checks for a period and for the Control, Option, and Command keys. If only the Command key has been pressed, item 2 is returned as being hit. Similarly, the next item-match string handles Command-Escape (Escape is 1B) and the last item-match string handles Escape by itself.

main DITL and uses an 'hdlg' resource and an STR# resource with the same ID. The second help item is an 'hdlg' resource for each pane's DITL; it should start at item 8 for the first control in the pane. See the file MPDialogs.µ.rsrc on the CD for a sample 'hdlg' resource for the first pane.

CALLING THE MAIN ROUTINES

Now we'll review the calls your application needs to make to the main routines in order to open and close the multipane dialog, handle events, and access the values of the controls in the dialog. But first, let's look at the data your application needs to maintain.

POINTERS AND HANDLES

Your application must maintain a DialogPtr for each dialog used. You also need to declare a handle for storing the returned settings. Passing a pointer to NULL causes the code to allocate a new handle and return it to the caller; otherwise, a handle to an existing record must be provided. For a preferences dialog, this data should be maintained in the application's preferences file in the Preferences folder.

> **Implementing preferences files** is discussed in the article "The Right Way to Implement Preferences Files" in *develop* Issue 18.•

The sample code internally allocates an MPDHdl for each open multipane dialog for storing state information. The handle is stored in the refCon of the dialog.

OPENING, HANDLING EVENTS, AND CLOSING

Your application should call OpenMPDialog for each desired multipane dialog, taking any actions necessary when a dialog is opened, such as disabling menus. This call is passed the resource ID of the DLOG for the dialog, a reference to the handle that stores the returned settings, and four optional parameters, which are described later.

Here's an example:

```
DialogPtr prefDlog = NULL;
Handle thePrefs = NULL;

prefDlog = OpenMPDialog(kPrefDLOG, NULL, NULL, NULL, NULL, &thePrefs);
if (prefDlog) SetMenusBusy();  // If NULL, the dialog couldn't be opened.
```

The main event loop should call DoMPDialogEvent after each event is returned from WaitNextEvent. If DoMPDialogEvent returns true, the multipane dialog routines have handled the event; your application should inspect the DialogPtr to determine whether the dialog has been closed, so that the application can recover from the dialog state. A return value of false indicates that your application should process the event as it would normally. For example:

```
if (DoMPDialogEvent(&prefDlog, &mainEventRec)) {
   // A NULL DialogPtr means the dialog has been closed.
   if (!prefDlog)
      SetMenusIdle();
} else {
   // Process the event as usual.
   ...
}
```

To dispose of the dialog without user interaction, your application can call CloseMPDialog:

```
CloseMPDialog(prefDlog);
```

After the dialog has been closed, it's the application's responsibility to dispose of or save the data handle created with the call to OpenMPDialog. The code I've provided assumes this handle is maintained by the application after creation.

ACCESSING CONTROL VALUES

The following two routines are provided for accessing the control values stored in the data handle:

- GetMPDItem retrieves the value of the control corresponding to the pane and item specified and stores it in a buffer.
- SetMPDItem stores in the handle a value retrieved from a buffer.

Both of these routines assume that the caller knows the length and type of the control's data representation. Items are numbered differently from in the DITL resource — only items that have a value are included, and the values for radio button groups come after those for all other controls in the data. The values of checkboxes, enabled buttons in radio button groups, and pop-up menus are stored as 16-bit integers. Return codes are defined in the header file. Errors are returned for invalid pane and item numbers and buffer lengths.

The routines are declared as follows:

```
short GetMPDItem(Handle theData, short pane, short item, Ptr ptr, short len)

short SetMPDItem(Handle theData, short pane, short item, Ptr ptr, short len)
```

The sample application, in the code for DialogDisplay, provides a basic example of the use of these routines to display the current settings of the controls in the previously closed dialog.

Normally, these routines should be sufficient to access the data in the handle. However, those applications for which it would be more efficient to manipulate the handle directly can use the following format:

```
Last Open Pane
Offset to Pane 1, Offset to Pane 2, ..., Offset to Pane n, NULL
(Pane 1) Length of Item 1, Data for Item 1, ..., Length of Item m, Data
   for Item m, NULL
...
(Pane n) Length of Item 1, Data for Item 1, ..., Length of Item m, Data
   for Item m, NULL
```

The Last Open Pane and the Offset to Pane fields are all long integers and the Length of Item fields are all short integers. The Length of Item value doesn't include the length of itself; to get to the next field you would add

```
Length of Item + sizeof(short)
```

to the pointer. The Last Open Pane field allows the multipane dialog code to display the dialog with the last pane the user had open as the current pane.

That's all you need to know to make basic use of my multipane dialog code. But you can also go a step further: you can customize certain aspects of a multipane dialog by using the four optional parameters to OpenMPDialog mentioned above.

CUSTOMIZING WITH OPTIONAL PROCEDURES

The second through fifth parameters to OpenMPDialog can indicate action procedures that customize dialog behavior by responding to certain events. A value of NULL for any of these parameters tells the application to use the default behavior. To provide custom behavior, you would pass a universal procedure pointer instead of NULL. The procedures can also be changed dynamically, with the InstallAction routine.

The action procedures and the default actions are as follows:

- The Set Defaults action procedure (parameter 2) provides factory defaults for controls. The default action is to set them to 0.
- The Click action procedure (parameter 3) enables you to customize the actions resulting from clicking a control, such as dimming or undimming other controls or performing data validation. The default action is to toggle checkboxes and handle radio buttons via the Radio Group action procedure.
- The Edit action procedure (parameter 4) enables special handling of editable text fields, such as converting the string to an integer. The default action is to store the entire string as a Str255.
- The Radio Group action procedure (parameter 5) enables you to customize the behavior of radio button groups, such as how the values are stored. The default action is to store the value as the index number of the radio button that's enabled in the group; the default value is 1 (the first radio button in the group).

All the action procedure pointers are declared as UniversalProcPtrs for compatibility in case of PowerPC compilation, so they must be allocated before use. The sample program does this by declaring a UniversalProcPtr for each desired action procedure. For example, the one for the Click action procedure is declared as follows:

```
ClickActionUPP myClickAction = NULL;
```

It's initialized in the main routine of the application like this:

```
myClickAction = NewClickActionProc(MyClickAction);
```

Depending on what you want to do in the action procedures, you may need to make use of the MPDHdl stored in the dialog's refCon, mentioned earlier. This is a handle to an MPDRec (shown in Listing 1), which is the main data structure used by the multipane dialog code for state information. None of the elements of this structure should be modified by user code. The four UPP fields can be manipulated via calls to InstallAction and RemoveAction.

The **baseItems** field will be the most useful in the action procedures. It holds the item number of the first item in the pane, which is the hidden static text item used for Command-key equivalents. Thus, if dataH is of type MPDHdl, the index of the first real control (the second DITL entry) in the pane will be (*dataH)->baseItems + 1.

Now let's take a closer look at each of the action procedures.

Listing 1. The MPDRec structure

```
typedef struct MPDRec {
   short           numPanes;      // Number of panes in the dialog
   short           currentPane;   // Current pane being displayed
   short           baseItems;     // Item number of first item in panes
   short           *paneIDs;      // List of IDs for the pane's DITLs
   short           paneDirty;     // Whether Revert should be enabled
   RadioGroupPt    radio;         // Linked list of radio button groups
   Handle          theData;       // Actual storage for dialog values
   Handle          tmpData;       // Temporary storage for dialog values
   Handle          *IconHandles;  // List of icon suites
   ListHandle      theList;       // List Manager list for the icon list
   ClickActionUPP  ClickAction;   // Action procedures
   EditActionUPP   EditAction;
   GroupActionUPP  GroupAction;
   DefActionUPP    DefAction;
} MPDRec, *MPDPtr, **MPDHdl;
```

THE SET DEFAULTS ACTION PROCEDURE

The Set Defaults action procedure provides factory defaults for checkboxes and other controls, except for radio button groups (handled in the Radio Group action procedure). It's called with a pointer to — and the length of — a buffer holding the internal representation of the value of a single control corresponding to a specific pane and item number. You can call DefaultAction to take the default action for items your code doesn't handle.

The procedure is declared like this:

```
void MySetDefAction(Ptr theData, short len, short iType, short pane,
   short item)
```

The Set Defaults action procedure's defaults for radio buttons apply only to those that aren't part of a radio button group. But using single radio buttons is definitely not advised; all radio buttons should be in groups to be consistent with the *Macintosh Human Interface Guidelines*.

THE CLICK ACTION PROCEDURE

The Click action procedure enables you to customize the actions resulting from clicking a control. For instance, this procedure can handle dimming or undimming other items when certain controls are clicked. It can also provide validation for control settings when the user tries to change the pane or click OK, to ensure that the entered settings make sense.

The procedure receives a DialogPtr and the pane and item numbers. It's declared as follows:

```
short MyClickAction(short mType, DialogPtr dlog, short pane, short item)
```

The mType parameter specifies the message to process when the action procedure is called. The procedure is called with a kInitAction message right after the control is set when the pane is first displayed; this gives you an opportunity to set up the initial state of the dialog. The procedure is called with a kClickAction message after the user

has released the mouse button in a control. A kValidateAction message is received for data validation; it's the responsibility of the Click action procedure to put up an alert to notify the user if a setting is unacceptable.

Listing 2 is a Click action procedure from the sample application that undims the third checkbox in the Defense pane (Self-Destruct) if the second checkbox (Enable Self-Destruct) is checked. It also ensures that the editable text field in the Communications pane contains only digits; if this field contains nondigits, the validation fails and the user can't change panes or click OK.

The default Click action procedure, DefaultClickAction, calls the Radio Group action procedure to handle buttons in a radio button group; thus, actions in response to a click in a radio button group should be handled there. Call DefaultClickAction to inherit default functionality for controls not handled in your customization procedure.

Listing 2. A sample Click action procedure

```
short MyClickAction(short mType, DialogPtr dlog, short pane, short item)
{
   MPDHdl   dataH;
   short    iType, val = 0;
   Rect     iRect;
   Handle   iHandle;

   // Obtain multipane dialog state record.
   dataH = (MPDHdl) GetWRefCon(dlog);

   // Handle the second item validation.
   if (mType == kValidateAction) {
      // Validation fails if nondigits are in the field.
      if (pane == kCommPane &&
            item == kFrequency + (*dataH)->baseItems) {
         GetDialogItem(dlog, item, &iType, &iHandle, &iRect);
         GetDialogItemText(iHandle, theStr);
         val = VerifyDigits(theStr);
         if (val)
            StopAlert(ALERT_Invalid, NULL);
      }
      // All other items validate OK.
      return val;
   }

   // If this isn't the second checkbox, handle things the default way.
   if (pane != kMiscellaneousPane ||
         item != kEnableSelfDestruct + (*dataH)->baseItems)
      return (DefaultClickAction(mType, dlog, pane, item));

   // Initialize and Click messages are handled almost the same.
   // Dim the third checkbox based on the value of the second.
   GetDialogItem(dlog, item, &iType, &iHandle, &iRect);
   val = GetControlValue((ControlHandle) iHandle);
```

(continued on next page)

Listing 2. A sample Click action procedure *(continued)*

```
    switch (mType) {
        // Toggle the item in response to the user click.
        case kClickAction:
            val = !val;
            SetControlValue((ControlHandle) iHandle, val);
            // Fall through!
        // In either case, enable/disable next checkbox.
        case kInitAction:
            AbleDItem(dlog, kSelfDestruct + (*dataH)->baseItems, val);
            break;
    }

    // Initialize and Click messages should never fail.
    return 0;
}
```

THE EDIT ACTION PROCEDURE

The Edit action procedure enables special handling of editable text fields. A common implementation is to store the field's string as a long integer, converting the string value to and from this form as needed.

This procedure receives a pointer to a buffer for storage of the control's internal value, a handle to the control, and the pane and item numbers; it returns the length of the space required for the text field. The first parameter is a message that informs the procedure whether to calculate the storage size for this field, initialize the value, or copy the value to or from the field.

The procedure is declared as follows:

```
short MyEditAction(short mType, Ptr hPtr, Handle iHandle, short pane,
    short item)
```

The kCalcAction message requests the amount of storage required for the representation of the field value in memory. The kInitAction message requests that the value of the field be initialized. The kP2TAction message requests that the code retrieve the value of the field and store it in memory (in other words, that the permanent storage value be transferred to the temporary storage area — P2T is shorthand for "permanent to temporary"). Conversely, the kT2PAction message ("temporary to permanent") requests that the code set the field to the value indicated by the representation in memory. Default behavior can be maintained by calling DefaultEditAction, if desired.

Listing 3 is an Edit action procedure from our sample application. Normally, the procedure should check the item and pane numbers to distinguish between different text fields, but the sample application has only one such field.

THE RADIO GROUP ACTION PROCEDURE

To simplify using radio button groups, a single value is stored for the entire group. This value is the relative item number of the enabled button in the group. For example, the value of a group of three radio buttons with the second one enabled would be 2.

Listing 3. A sample Edit action procedure

```
short MyEditAction(short mType, Ptr hPtr, Handle iHandle, short pane,
   short item)
{
   short    ret = 0;
   long     val;
   Str255   textStr;

   Assert(hPtr != NULL);
   switch (mType) {
      case kP2TAction:     // Save value of control.
         GetItemDialogText(iHandle, textStr);
         StringToNum(textStr, &val);
         *(long *) hPtr = val;
         ret = sizeof(long);
         break;
      case kT2PAction:     // Set value of control.
         val = *(long *) hPtr;
         NumToString(val, textStr);
         SetIText(iHandle, textStr);
         ret = sizeof(long);
         break;
      case kInitAction:    // Initialize value.
         *(long *) hPtr = 0;
         ret = sizeof(long);
         break;
      case kCalcAction:    // How much storage do we need for this?
         ret = sizeof(long);
         break;
   }
   return ret;
}
```

In the sample program, radio button groups are stored in a linked list starting from the **radio** field of the MPDRec structure. The RadioGroup structure is defined as shown in Listing 4.

Listing 4. The RadioGroup structure

```
typedef struct RadioGroup {
   struct RadioGroup *next;
   short pane;
   short num;
   short items[1];
} RadioGroup, *RadioGroupPtr;
```

The **next** field points to the next radio button group, to enable traversing the linked list of groups. The **pane** field is the pane number this group belongs to. The **num** field holds the number of items that make up this radio button group. The relative item numbers of these radio buttons are stored in the **items** array.

The Radio Group action procedure enables you to customize the behavior of radio button groups. For instance, an application could choose to store radio button group values differently from the default or handle dimming or undimming of items in response to the user's actions. The Radio Group action procedure receives the same messages as the Edit action procedure. It returns the length of the space required for the radio button group's internal storage; the default is four bytes per group, two for the number of radio buttons and two for the value as a short integer.

Like the Edit action procedure, the Radio Group action procedure is called with the kInitAction and kCalcAction messages. However, these messages occur before the dialog is opened, so the DialogPtr will be NULL at that time. The procedure is declared like this:

```
short MyGroupAction(short mType, RadioGroupPtr group, Handle dataH,
   DialogPtr dlog, Ptr hPtr, short pane, short item)
```

Note that in response to the kInitAction message, the action procedure is expected to store the number of radio buttons in the group in the first two bytes of the internal storage. Here's an example from the default Radio Group action procedure (dataH is of type MPDHdl):

```
for (i = 0; i < group->num; i++) {
   if (GetCheckOrRadio(dlog, group->items[i] + (*dataH)->baseItems - 1))
      *(short *) hPtr = i + 1;
}
```

To obtain the actual item number for the control in the dialog, you just add

```
(*dataH)->baseItems - 1
```

to the relative number stored in the **items** array, as shown in the above code. As mentioned earlier, the **baseItems** field of dataH is the number of the first pane-specific item in the dialog.

NOW WHAT?

The code that accompanies this article on this issue's CD provides an easy-to-implement method for adding icon-selected multipane dialogs to any application. (The routines for managing radio button groups could be extracted without much difficulty and used elsewhere.) The sample program also provides an example of using the AppendDITL and ShortenDITL routines. So experiment with the sample application and then try out multipane dialogs as a way of simplifying the user interface in your own application.

Thanks to our technical reviewers Tim Craycroft, Nitin Ganatra, C. K. Haun, and Elizabeth Moller. Thanks also to Eric Soldan for ListControl and KeyEquivFilter from DTS Lib. •

CAL SIMONE

ACCORDING TO SCRIPT

Thinking About Dictionaries

I've been thinking lately about the purpose of this column, which debuted in the previous issue of *develop*. Permit me to take a moment to say something about that before I get down to some tips about dictionaries.

During the first couple of years after the birth of the Macintosh, there was a period of chaos, when application developers were figuring out how to extend the basic user interface. For example, some of the most commonly used menu commands appeared in different locations in various applications, and, more important, keyboard shortcuts varied or sometimes weren't present at all. After a while, though, things settled down and almost everyone adopted the standards that were eventually documented in the *Macintosh Human Interface Guidelines*.

AppleScript is the alternate user interface to your application. Now that AppleScript has been available for two years, it's time to move out of the "free-for-all" and develop the same consistency we've all come to enjoy and expect from the Macintosh experience. That's what this column (and the work I do in the AppleScript development community) is all about — encouraging consistency. The tips I offer here reflect undocumented conventions followed by many developers I've worked with, as well as my own thinking about scriptability. Until the time when standards are documented in a "Macintosh Human Scriptability Guidelines," I encourage you to adopt the techniques suggested here.

Though I've said it before, I'll say it one more time: adopting the object model is the single most important factor contributing to consistency in the AppleScript language across applications of different types. One developer I know resists using the object model year after year, arguing that it "isn't appropriate for everything." But the fact is that the object model has been successfully applied to a whole range of applications. Every major C++ framework now supports it or has add-ons to support it, and up-and-coming languages will support it. Even if your application has only one object (such as the dictionary of a small paging program I've seen), just do it!

ORGANIZING YOUR DICTIONARY

So far in the scripting world, various developers have used different schemes in their dictionaries for organizing the events in a suite, the parameters in an event, the properties in an object, and so forth. Some organize them according to their function, others order them alphabetically, and still others don't seem to have any scheme whatsoever (probably because scripting support was added a bit at a time or as an afterthought). For the sake of consistency across different scriptable applications, using some standard scheme is preferable.

If you're including an entire standard suite (such as the Core suite) from AppleScript's system dictionary (listed in the Rez files named EnglishTerminology.r, FrenchTerminology.r, and so on) and then overriding or extending the suite to add your own terms, make sure that your overrides appear in the same order as they do in the system dictionary and that extensions come after all the overrides. If you're implementing your own terminology, either as extensions to existing suites or in your own suites, organize it as described in the following paragraphs.

When you're adding new terms to a previously created dictionary (for example, when upgrading your application to provide deeper scripting support), remember to insert the new terms according to the same scheme or schemes you originally implemented. It's a good idea to keep some notes in your internal design documents describing the ordering schemes you used, so that you can be consistent with your earlier work (unless you're redoing your scripting implementation from scratch — for instance, when you're converting from an old non–object model implementation to the object model).

CAL SIMONE (AppleLink MAIN.EVENT) works way too hard at Main Event Software in Washington DC. He took his last summer vacation five years ago; it's been so long, he's forgotten what a vacation is like, and he can't imagine where he'd go. He's been to beautiful mountainous places like Colorado, Alaska, British Columbia, and Switzerland, and to a few islands like Saint Thomas and the Bahamas. Cal would really like to hear your suggestions on possible future topics for this column, as well as your ideas for good vacation spots.

Suites. So that your dictionary is consistent with dictionaries in other applications, include the standard Registry suites first (Required suite first, then the Core suite, then any other Registry suites). Then include any custom suites you create.

Events. Order commands that correspond to events in one of four ways: by likelihood of use, according to function, chronologically, or alphabetically. The method you choose will depend on how your application is used and the nature of your users. As an example of each of these schemes, I'll show how some of the Core suite verbs might be organized.

If certain commands are to be used more frequently than others, order them according to likelihood of use. Present those commands that will be used most frequently at the beginning and those seldom used at the end:

get	(more of these than anything else)
set	(quite a few of these, too)
count	(a fair amount of counting)
make	(sometimes new objects are created)
open	(sometimes they're opened)
close	(and closed)
print	(printing isn't done as frequently)
delete	(neither is deleting)
quit	(quitting is done only occasionally)

If your users will logically group the operations, use an ordering according to function. Group together commands that are related in some way:

make	(make and delete)
delete	
open	(open and close)
close	
set	(set and get)
get	
count	(the rest are unrelated)
print	
quit	

If the commands are normally used in a certain order, choose a chronological ordering. First present the commands that will be used first, followed by the commands that will be used later:

make	(this often comes first)
open	(or else opening comes first)
set	(then setting properties)
get	(and later getting properties)
count	(counting comes in the middle)
print	(printing happens later)
close	(then comes closing)
delete	(deleting is near the end)
quit	(last, we bail out)

If the commands aren't going to be used in any particular order, or you don't know what that order is likely to be, and there's no logical grouping, list the commands alphabetically, as the Core suite does. Although alphabetical order isn't as helpful as the other schemes, script writers will at least be able to find commands more easily in your application's dictionary.

Parameters. Make an effort to list parameters in an order that encourages the writing of natural, grammatically correct sentences for commands. For example:

```
make  new  <type class>
      [at  <location reference>]
      [with data  <anything>]
      [with properties  <record>]
```

If the order of an event's parameters doesn't matter as far as sentence style is concerned, order them according to the frequency of likely use.

```
close  <reference>
       saving  <yes|no|ask>
       saving in  <file specification>
```

Object classes and properties. I'd suggest placing the outermost objects in your containment hierarchy first, objects contained in the outermost objects next, and objects that don't contain any other objects last. Remember that every object class representing an actual object must be listed as an element of some other object, eventually leading back to the application class (the null container). Primitive class definitions and record definitions (which aren't part of the containment hierarchy) and abstract classes (which aren't instantiable objects but are used to hold lists of inherited properties) should be placed in the Type Definitions or Type Names suite, and clearly labeled as a record definition or abstract class. (See my article, "Designing a Scripting Implementation," in *develop* Issue 21.)

Properties of objects can be ordered according to one of the schemes described above for events.

WHEN YOU ALLOW MULTIPLE VALUE TYPES

Occasionally in your dictionary you might need to specify a parameter or property for which any of several types is acceptable. Using the wild card ('****') as the type of a parameter or property tells your user that you'll accept *anything* (or at least a wide variety of mixed types). Don't do this to be lazy or to finish your dictionary quickly; do it only if you mean it. If you

accept only one type, explicitly indicate so. If you allow two different types, you can either create a compound "type" or use identical keyword entries.

Defining a compound "type." One way of handling cases where you can accept two different value types for a parameter or property is to make up a new "type" to represent a combination of acceptable types in your dictionary. This isn't a real type that you'd have to check for or deal with in your application's code, but instead just serves to indicate in your dictionary that your application will handle *either* type. This works particularly well when the value types are simple. For example:

```
class reference or string: Either a reference or
   a name can be used.
```

You can use your new "type" in a parameter or property definition as follows:

```
class connection
properties:
   window  <reference or string>  -- the
      connection's window can be referred to
      either by a reference or by its name
```

To define a new type, make a new object class and place it in the Type Names suite (see my article in Issue 21).

Using identical keyword entries. You can also use multiple entries with identical keywords to specify alternative ways of filling in a parameter or property value. This works well when the value types are complex or are highly dissimilar. For example, the **display dialog** command has two **with icon** listings, one for specifying the icon by its resource name or ID and the other for displaying the stop, note, or caution icon:

```
display dialog  <anything>  -- title of dialog
   ...  other parameters
   [with icon  <anything>]  -- name or id of the
      icon to display
   [with icon  <stop|note|caution>]  -- or display
      one of these system icons
```

Note the use of "or" in the second entry's comment: make sure you use the same 4-byte ID for *both* parameter entries.

Although you could have many entries to show every possible individual type that a parameter or property takes, this might become confusing to the user. So I'd recommend that you use this sparingly, and when you do use it, try to limit the number of similar entries to 2.

MAKING USE OF THE COMMENT AREA

You can use the comment area (available for each suite, event, parameter, class, and property entry) to help clarify how your vocabulary is to be used. Since your dictionary is often the initial "window" through which a user looks to figure out what to do, descriptive comments can make the user's task a lot easier. And remember that your users aren't necessarily programmers, so you should avoid terms like FSSpec in your comments. I'll give some examples to show you what I mean.

- For Boolean parameters and properties, if there are two possible states, include a description of the true and false conditions, such as "true if the script will modify the original images, false if the images will be left alone."
- If the possible states are on and off, you need only include the true condition ("If true, then screen refresh is turned on") or ask a question ("Is the window zoomed?").
- For enumerations, include a general description of what the parameter or property represents; the individual enumerators should be self-explanatory. For example, "yes | no | ask -- Specifies whether or not changes should be saved before closing."
- Don't use the comment field to explain a set of possible numeric values when an enumeration (with descriptive enumerators) is better. Instead of "0=read, 1=unread, ..." use "read | unread | ..."
- For compound "types," describe the parameter or property, as well as the choices for value types listed: "the connection's window (either a reference or name can be used)."
- For "anything" (unless you actually allow *any type* the user can think of), describe which specific types you allow: "[... descriptive info] (a string, file reference, alias, or list is allowed)."
- If you allow either a single item or a list, indicate so: "the file or list of files to open."
- If the parameter or property has a default value (used when the user doesn't include an optional parameter or set the property), mention it (this applies to values of any type): "replacing yes | no | ask -- Replace the file if it exists? (defaults to ask)."

Keep in mind that if you include an entire standard suite (such as the Core or Text suite), your own comments should reflect the style of the comments in that suite. See the Scriptable Text Editor's dictionary as an example of fairly good comment style; it shows the standard versions of the Required, Core, and Text suites and adds some of its own terminology.

A COUPLE MORE DICTIONARY TIPS

While I'm on the subject of dictionaries, here are a couple of extra tidbits.

Use only letters and numbers for terms in dictionaries. Don't use a hyphen (-), a slash (/), or any other nonalphanumeric characters in your dictionary entries. For example, if you use **Swiss-German**, AppleScript will treat it as **Swiss - German** (subtraction), which is not what you want; if you use **Read/write**, it will be treated as **Read / write** (division). Note that **Read/write** is in the standard Table suite, but it won't compile properly.

All terms must start with letters. Using **9600** as an enumerator won't work; you would have to use something like **baud9600**.

Finally, pick names for your terms that are descriptive for a user, especially a nonprogrammer. If you pick a term like **x**, users won't be allowed to use **x** as a variable name in their scripts. For instance, instead of "x <small integer> -- the x coordinate" use "horizontal coordinate <small integer> -- the x coordinate."

IT'S YOUR THING

Unlike writing code, designing a scripting vocabulary isn't an exact science. It's up to you to decide in what manner (and how effectively) humans will interact with this new interface. Applying "programming language" concepts and standards won't always work. You need to keep an eye toward the human aspects of the AppleScript language and to work out a scheme that reflects careful attention to your users.

You may occasionally see guidelines here that aren't completely clear-cut or that even conflict with each other, and every so often I'll adjust what I've said in an earlier column. This is the nature of an evolving language. If you're not completely at home with this, seek out an expert in scriptability design for advice. But remember, vocabulary design is by nature as much art as science.

Thanks to Sue Dumont and C. K. Haun for reviewing this column. •

How're we doing?

If you have questions, suggestions, or even gripes about *develop*, please don't keep them to yourself. Drop us a line and let us know what you think.

Send editorial suggestions or comments to AppleLink DEVELOP or to:

Caroline Rose
Apple Computer, Inc.
1 Infinite Loop, M/S 303-4DP
Cupertino, CA 95014
AppleLink: CROSE
Internet: crose@applelink.apple.com
Fax: (408)974-6395

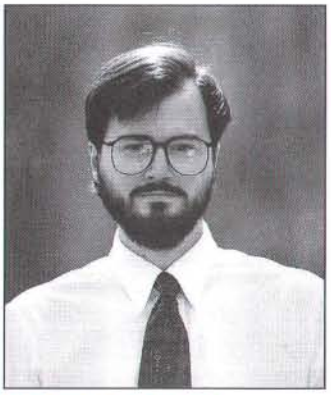
Send technical questions about *develop* to:

Dave Johnson
Apple Computer, Inc.
1 Infinite Loop, M/S 303-4DP
Cupertino, CA 95014
AppleLink: JOHNSON.DK
Internet: dkj@apple.com
CompuServe: 75300,715
Fax: (408)974-6395

Please direct all subscription-related queries to *develop*, P.O. Box 319, Buffalo, NY 14207-0319 or AppleLink APDA (on the Internet, apda@applelink.apple.com). You can also call 1-800-282-2732 in the U.S., 1-800-637-0029 in Canada, or (716) 871-6555 from all other locations.

Document Synchronization and Other Human Interface Issues

One of the things the Finder does best is maintain the illusion that an icon and its window represent a single object. Using the routines described in this article, your application can help maintain that illusion. You can ensure that when the user renames an open document, the change is reflected in the document window's title. You can also gracefully handle problems that may arise if the document file is moved. Other improvements that make your application's interface more consistent with the Finder's include preventing a second window from opening when an open document's icon is double-clicked and adding a pop-up navigation menu to the document window's title bar.



MARK H. LINTON

To rename a folder or file in the Finder, you click the icon name, type a new name, and press Return. For folders, if the window is open, the change is reflected right away in the window's title bar. But for files, if the document is open in your application, its window may not reflect the name change. Try this little experiment: Create a document in your application and save it. Switch to the Finder, find your document, and change its name. What did your application do? If it's like most applications, nothing happened: the document window has the same name as before. Go ahead and try to use Save As to give the file the same name you gave it in the Finder. You probably get an error message. Now try to save the document under the original name. Do you still get an error message? Quit your application and read on for a way out of this frustration.

The only convenient way for a user to rename a document is with the Finder. (The Save As command doesn't rename a document; it creates a copy of the document with a new name.) As you've just seen, name changes made in the Finder aren't automatically reflected in an open document window. Another change that's often not picked up by the application is when the user moves the document to a different folder. The code in this article helps synchronize your application's documents with their corresponding files, so that a document will respond to changes made outside the application to its file's name or location.

This article also describes how to prevent a duplicate window from being opened if the user opens an already open document in the Finder and how to add a pop-up menu to the document title bar to help the user determine where the file is stored. All

MARK H. LINTON (mhl@hrb.com) lives in Centre Hall, Pennsylvania, with his wife Gretchen. When he isn't jetting around the globe or meeting with some high government officials as part of his job as senior engineer at HRB Systems, he can be found in his log cabin at the base of Mount Nittany playing with his Macintosh.

the code for implementing these features is provided on this issue's CD, along with a sample application that illustrates its use.

DOCUMENT SYNCHRONIZATION

The *Electronic Guide to Macintosh Human Interface Design* says that applications should "match the window title to the filename." Specifically, when a user changes the document name in the Finder, you should update all references to the title. The guide also refers to the *Macintosh Human Interface Guidelines*, page 143, where it says, "The document and its corresponding window name must match at all times."

When I first started looking at the problem of document synchronization, I assumed that the animated example in the *Electronic Guide to Macintosh Human Interface Design* was the way to go. In this animation, the application checks for a name change when it receives a resume event. However, I became uncomfortable with this approach, because it would cause a delay between the user's changing the name of the document in the Finder and the application's updating the window title. Using a resume event relies on a separate action by the user, namely, bringing the application to the foreground. This seemed nonintuitive and didn't support the illusion that a window and its icon represent a single object. Also, it's possible that with Apple events and AppleScript an application could be launched, do some work, and quit without ever being frontmost — that is, without ever receiving a resume event.

The truth is that these days, with multiple applications running at the same time, with networked, shared disks everywhere, and with applications and scripts pulling the puppet strings as often as users, a file's name or location may change at any time, whether the application is in the foreground or the background. A script might move or rename a file or, if the file is on a shared volume, another user on the network could move or rename it or even put the file in the Trash — all behind the application's back. The only solution I found under the current system software was to regularly look at the file to see if its name or location has changed. In other words, the application has to poll for changes.

Polling is generally a bad idea, but there are cases when it's the only reasonable way to accomplish a task, and this is one of them. However, I tried to keep the polling very "lightweight" and low impact by using the following guidelines:

- An application shouldn't poll any more often than it absolutely needs to. Waking up an application causes a context switch, and context switches take a significant amount of time. Forcing the system to wake up an application every few ticks just so that it can look for file changes would be a bad idea, especially when the application is in the background. Instead, the application should poll only when an event has already been received — that is, when the application is awake. Set your WaitNextEvent sleep time appropriately, and wait at least a second or two between "peeks." (The Finder, for instance, polls for disk changes every five seconds or so.)
- Avoid any polling that causes disk or network access; if at all possible, examine only information that's in RAM on the local machine. Network access in particular can be a real drain on performance.

The sample code follows this advice, doing everything it can to be unobtrusive. It polls for file changes only once every second while in the foreground. In the background, the application's WaitNextEvent sleep time is set to ten seconds, so it only wakes up — and thus polls — every ten seconds if nothing else is going on. To detect changes to files, I chose to examine the volume modification date of the volume containing the file, since this information is always available in local RAM,

even for a shared volume. If that date changes, I look deeper to see if the change is one I'm interested in. As you dip into the code, you'll see the details.

I use the file reference number to track files because it survives changes in the name and parent directory. However, this requires that the files be kept open. If you can't keep your files open, you might want to look at John Norstad's excellent NewsWatcher application, which uses alias records to synchronize files. NewsWatcher is on this issue's CD; its official source can be found at ftp://ftp.acns.nwu.edu/pub/newswatcher/.•

Friendly as it is, this polling solution is appropriate only for the current system software; future system software versions (such as Copland, the next generation of the Mac OS) will provide a much better way to detect changes. Your application will be able to subscribe to notification of changes that it's interested in. In fact, polling the current file system structures will be unfriendly behavior under Copland, which will have demand-paged virtual memory and a completely new file system. For this reason, the sample code is designed to work only under System 7. You'll be able to easily retrofit the code to run under Copland once the details of the correct way to detect file changes have been worked out.

THE HEART OF THE MATTER

Every Macintosh programmer eventually comes to grips with how to keep track of all the information associated with a document. I use a structure called a *document list* and I have a set of routines that support it. The document list reverses some common assumptions used by developers. Developers often use the window list to track their windows and attach their document data to it, but this limits Apple's ability to redefine the window list. My recommendation is to create a document list (almost identical to the window list) containing the document data and attach the windows to it. In this way, the actual *structure* of the window list is not a concern. You'll find my implementation of the document list and its supporting routines on this issue's CD.

While the code presented here is specific to my implementation, you can easily generalize it as needed. The code below shows how your application might call DSSyncWindowsWithFiles, a routine that keeps your documents synchronized with the Finder by checking for and handling changes made outside the application to file names or locations. Call the routine from within your main event loop when you receive an event (including null events). Note that error checking has been removed from the code shown in the article, but it does appear on the CD.

```
while (!done) {
   gotEvent = WaitNextEvent(everyEvent, &theEvent, gSleepTime,
      theCursorRegion);
   if (gotEvent)
      DoEvent(&theEvent);
   DSSyncWindowsWithFiles(kDontForceSynchronization);
}
```

This minor change does most of the work for your application. The machinery that makes it happen lies within DSSyncWindowsWithFiles (see Listing 1). This routine first checks to make sure that enough time has passed since the last check for changes. If so, or if the caller requested immediate synchronization, it iterates through each of the windows registered in the document list, calling DSSyncWindowWithFile to process each of these windows.

DSSyncWindowWithFile, shown in Listing 2, begins by getting the file reference number for the window from the document list. If it's appropriate to continue

Listing 1. DSSyncWindowsWithFiles

```
#define kCheckTicks 60

pascal void DSSyncWindowsWithFiles(Boolean forceSync)
{
   WindowPtr      theWindow;
   static long    theTicksOfLastCheck = 0;
   long           theTicks;

   theTicks = TickCount();
   if (theTicks > (theTicksOfLastCheck + kCheckTicks) || forceSync) {
      theTicksOfLastCheck = theTicks;
      for (theWindow = DSFirstWindow(); theWindow != nil;
           theWindow = DSNextWindow(theWindow)) {
         DSSyncWindowWithFile(theWindow);
      }
   }
}
```

Listing 2. DSSyncWindowWithFile

```
pascal void DSSyncWindowWithFile(WindowPtr aWindow)
{
   short theFRefNum;

   DSGetWindowDFRefNum(aWindow, &theFRefNum);
   if (DoSyncChecks(theFRefNum, aWindow)) {
      HandleNameChange(theFRefNum, aWindow);
      HandleDirectoryChange(theFRefNum, aWindow);
      HandleMoveToTrash(theFRefNum, aWindow);
   }
}
```

(DoSyncChecks returns true), DSSyncWindowWithFile calls three other routines to handle name changes, changes that move the file to a different folder, and changes that move the file to the Trash.

THE CHECKPOINT

The DoSyncChecks routine (Listing 3) checks for changes to the volume that the file is on. If the volume has been modified, DoSyncChecks returns true to DSSyncWindowWithFile, which consequently calls the next three routines — HandleNameChange, HandleDirectoryChange, and HandleMoveToTrash.

A FILE BY ANY OTHER NAME

After determining that the volume containing the file has been modified, DSSyncWindowWithFile calls HandleNameChange (Listing 4). This simple routine compares the names of the window and the file; if they're not exactly the same, it updates the window to reflect the new filename. A minimal implementation of document synchronization might include only this routine.

Listing 3. DoSyncChecks

```
static Boolean DoSyncChecks(short aRefNum, WindowPtr aWindow)
{
   Boolean        doCheck = false;
   unsigned long  theLastDate, theDate;
   short          theVRefNum;

   if (aRefNum != 0) {
      DSGetWindowFileVRefNum(aWindow, &theVRefNum);
      GetVolumeModDate(theVRefNum, &theDate);
      DSGetWindowVLsBkUp(aWindow, &theLastDate);
      if (theLastDate != theDate) {
         DSSetWindowVLsBkUp(aWindow, theDate);
         doCheck = true;
      }
   }
   return doCheck;
}
```

Listing 4. HandleNameChange

```
void HandleNameChange(short aFRefNum, WindowPtr aWindow)
{
   Str255   theTitle, theName;

   GetWTitle(aWindow, theTitle);
   GetNameOfReferencedFile(aFRefNum, theName);
   if (!EqualString(theTitle, theName, true, true))
      SetWTitle(aWindow, theName);
}
```

Have you been wondering where the magical file management calls that DoSyncChecks and HandleNameChange use come from — for example, GetVolumeModDate and GetNameOfReferencedFile? See the file EvenMoreFiles.c on the CD for details. This is my tribute to Jim Luther's excellent MoreFiles collection. Whenever I need a routine that's not in the standard header, I write it and add it to the collection. Someday we'll be up to SonOfMoreFiles and NightOfTheLivingMoreFiles. •

MOVING TO A NEW NEIGHBORHOOD

After checking, and possibly synchronizing, the filename, DSSyncWindowWithFile calls HandleDirectoryChange (Listing 5) to see whether the file has been moved. This routine starts out by comparing the old parent directory to the new parent directory. If they're not the same, the file has been moved and the routine stores the file's new parent directory for later use by the application. It's possible that the file was moved to a parent for which the user doesn't have access privileges. In that case, a later Save will fail and revert to a Save As.

GETTING TRASHED

Finally, DSSyncWindowWithFile calls HandleMoveToTrash (Listing 6) to see if the file is in the Trash. If it is, HandleMoveToTrash gets the FSSpec corresponding to the file reference number, which will be needed later. If the application is running in

Listing 5. HandleDirectoryChange

```
void HandleDirectoryChange(short aFRefNum, WindowPtr aWindow)
{
   long     theOldParID, theNewParID;

   DSGetWindowFileParID(aWindow, &theOldParID);
   GetFileParID(aFRefNum, &theNewParID);
   if (theOldParID != theNewParID)
      DSSetWindowFileParID(aWindow, theNewParID);
}
```

Listing 6. HandleMoveToTrash

```
static void HandleMoveToTrash(short aFRefNum, WindowPtr aWindow,
      Boolean *inTrashCan)
{
   FSSpec         theFile;
   Boolean        inBackground;
   short          theResponse;
   EventRecord    theEvent;

   FileInTrashCan(aFRefNum, inTrashCan);
   if (*inTrashCan)
      GetFileSpec(aFRefNum, &theFile);
   if ((aFRefNum != 0) && *inTrashCan) {
      if (DSIsWindowDirty(aWindow)) {
         InBackground(&inBackground);
         if (inBackground) {
            DSNotify();
            do {
               InBackground(&inBackground);
               if (WaitNextEvent(everyEvent, &theEvent, gSleepTime, nil))
                  DoEvent(&theEvent);
               FileInTrashCan(aFRefNum, inTrashCan);
            } while (inBackground && *inTrashCan);
            DSRemoveNotice();
         }
         if (*inTrashCan) {
            ParamText(theFile.name, "\p", "\p", "\p");
            theResponse = Alert(rCloseAlert, nil);
            switch (theResponse) {

               case kSave:
                  DoSave(aWindow);
                  /* Fall through */

               case kDontSave:
                  ZoomWindowToTrash(aWindow);
                  DoCloseCommand(aWindow);
                  break;
```

(continued on next page)

Listing 6. HandleMoveToTrash *(continued)*

```
                case kPutAway:
                   DSAESendFinderFS(kAEFinderSuite, kAEPutAway, &theFile);
                   *inTrashCan = false;
                   break;
             }
          }
       } else   /* Window is clean; just close it */
          DoCloseCommand(aWindow);
   }
}
```

the background, and there are unsaved changes to the document, the routine notifies the user (with the Notification Manager) that the application needs assistance. While waiting for the user to respond to the request for assistance, HandleMoveToTrash handles events normally and also checks to see whether the user has moved the document back out of the Trash. After all, there's no sense in asking the user what to do about a file in the Trash if it's no longer there. If the user responds to the request, or moves the file out of the Trash while the application is still in the background, HandleMoveToTrash removes the notification. If the file is still in the Trash when the application becomes frontmost, an alert appears asking the user what to do.

Now if this were the Finder, there would be no question of what to do in this situation. When the user drags the icon for a folder to the Trash, the folder is essentially gone, so the associated window doesn't remain on the desktop. In the application world, life is a little more problematic. What happens if there are unsaved changes in the document? If the application blindly closes the document when the user drags the icon to the Trash, data could be lost. This would be a Bad Thing.

My mother always told me, "When in doubt, ask." So if there are unsaved changes to the file, an alert gives the user three choices: Don't Save, Remove From Trash, and Save. The Save and Don't Save options are simple: each closes the window as expected. Remove From Trash is a little tricky and takes advantage of the Scriptable Finder and Apple events.

The Remove From Trash case is similar to the Finder situation in which the user decides not to throw the document in the Trash and chooses Put Away from the File menu. HandleMoveToTrash handles this change of mind the same way the Finder handles it with Put Away: it sends the Finder a Put Away Apple event specifying the file in question as the target. (If the Scriptable Finder isn't available, the same action can be simulated manually; see the code on the CD for details.)

HOW CAN YOU BE IN TWO PLACES AT ONCE?

That's all there is to document synchronization. Now let's take a look at some other ways you can make your application's interface more consistent with the Finder's.

Many applications create a new window when an already open document is opened again in the Finder. But if the Finder were to open a second copy of a folder when you double-click the icon of a folder that's already open, wouldn't you be surprised? One of the guiding principles of human interface design is consistency; if your application doesn't perform the same action as the Finder (in this case, bring an already open window to the front), the user must learn and remember what will

happen in each particular situation. This detracts from the user's happiness with your application.

Making your application notice that the document is already open is easy if you're using the document list. The following code would appear where you normally call your open-file routine. When the application receives an event to open a file, it checks to see if the file is already registered in the document list. If it's registered, the application simply brings it to the front instead of opening it again.

```
if (DSFileInDocumentList(aFile, &theWindow))
   SelectWindow(theWindow);
else
   DoOpenFile(aFile);
```

POP-UP NAVIGATION

A nifty feature introduced with the System 7 Finder is the pop-up menu in the title bar that allows the user to determine the location of an open folder and to navigate the file system without having to resort to browsing (see Figure 1). The user simply holds down the Command key and presses on the window title to see the menu. The computer knows where your document is; it just needs a good way to present the information. If you have Metrowerks CodeWarrior, you'll find that it does something similar to the System 7 Finder. Your application can provide the same interface.

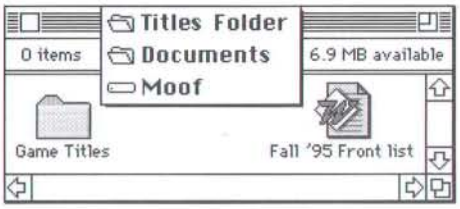


Figure 1. The Finder's pop-up navigation menu

To provide a pop-up navigation menu for your document windows, replace the existing call to FindWindow in your mouse-down event handler with a call to the DSFindWindow routine. DSFindWindow is simply a wrapper for the Window Manager's FindWindow routine. If FindWindow returns inDrag, DSFindWindow does some additional checking to determine whether the window is frontmost, the Command key is down, and the mouse is in the window title area. If the mouse-down event meets these conditions, DSFindWindow calls DSPopUpNavigation, which implements the menu and returns inDesk as the window part, telling the application to ignore the click.

Note that DSPopUpNavigation makes an assumption about the location of the window's title that may not be true for nonstandard window types or in future versions of the system software. In such cases the pop-up menu will still work fine, though it may not be cosmetically correct. This is another area of the code that should be revisited when Copland becomes available.

CONSISTENCY PAYS OFF

Consistency is one of the key principles that make using the Macintosh the wonderful experience that it is. If your program responds to the user's actions in the same way that the Finder does — in particular, maintaining the illusion that an icon and its window represent a single object — your users can explore your application with

skills they've already acquired. The techniques presented here show how to provide that extra measure of consistency with the Finder that keeps the Macintosh interface clean, consistent, and seamless. They're not too hard to implement, they're fun, and they just happen to be useful!

RECOMMENDED READING

- *Electronic Guide to Human Interface Design* (Addison-Wesley, 1994). This CD (available from APDA) combines the *Macintosh Human Interface Guidelines* and its companion CD, *Making It Macintosh*, into one easy-to-swallow capsule. Take one every night before going to bed, and wake up with a more consistent user interface.
- *Macintosh Human Interface Guidelines*, (Addison-Wesley, 1993). Available separately from APDA in book form.
- Polya, G., *How to Solve It* (Princeton University Press, 1945). This book explains a logical approach to problem solving. Very simply, the approach is: understand the problem, compare it to a related problem that has been solved before to arrive at a plan, carry out the plan, and examine the solution. That's what I've done with the subject of this article.

Thanks to our technical reviewers Jens Alfke, Greg Anderson, Arno Gourdol, Bill Keenan, Jim Luther, and Elizabeth Moller.•

Macintosh Q & A

Q *How do I create a menu with an icon as its title?*

A Set the menu title to 0x0501*handle*, where *handle* is the result of calling GetIconSuite. An example snippet of code follows; this code assumes that the menu title is already five bytes long.

```
void ChangeToIconMenu()
{
   Handle      theIconSuite = nil;
   MenuHandle  menuHandle;

   GetIconSuite(&theIconSuite, cIcon, svAllSmallData);
   if (theIconSuite) {
      menuHandle = GetMenuHandle(mIcon);
      if (menuHandle) {
         // Second byte must be 1, followed by the icon suite handle.
         (**menuHandle).menuData[1] = 0x01;
         *((long *)&((**menuHandle).menuData[2])) = (long)theIconSuite;
         // Update display (typically you do this on startup).
         DeleteMenu(mIcon);
         InsertMenu(menuHandle, 0);
         InvalMenuBar();
      }
   }
}
```

Q *We're drawing palette icons with a loop that consists basically of the following:*

```
GetIcon...
HLock...
CopyBits...
ReleaseResource...
```

Our native PowerPC version seems to draw these icons a lot more slowly than even our 680x0 version running under emulation, which suggests a Mixed Mode Manager slowdown. Which of these routines are currently native on the PowerPC? GetIcon and ReleaseResource together seem to take over 90% of the time.

A As you suspected, the Resource Manager calls you're using aren't native, and they generally call File Manager routines, which aren't native either. But be careful: what's native and what isn't is changing over time, and you shouldn't design your application based on today's assumptions.

That said, relying on the Resource Manager to be fast is generally not a good idea, native or not. One approach is to "cache" the icons, making sure they're in RAM at all times. (In general, you should do this for any user interface element that will be redrawn repeatedly while your application is open.) You can load your icons as one of the first few operations in your initialization code, just after calling MaxApplZone (possibly moving them high and locking them, since you don't want them to move during a CopyBits operation). This technique yields very good performance on the redraws that the palette needs, in exchange for a few kilobytes of memory. Don't forget to mark the resources as nonpurgeable.

Even better, if it will suit your purposes, would be to use the Icon Utilities to retrieve and draw your icons (as documented in *Inside Macintosh: More Macintosh*

Toolbox) and to build an icon cache. Using the Icon Utilities helps your application do the right thing for different screen depths. Also, the icon-drawing routines have been optimized to perform well under a variety of conditions.

Q *How can we detect that our application is already running and bring it to the front?*

A Simply iterate through the currently running processes with GetNextProcess, calling GetProcessInformation for each one and comparing its process signature with your application's (for an example, see the article "Scripting the Finder From Your Application" in *develop* Issue 20, page 67, Listing 1). If your application is running, call SetFrontProcess to bring it to the front.

Q *WindowShade is causing a problem for our application, which saves the window position and size when it saves a document to disk. If our application's windows are "rolled up" with WindowShade, its windows appear to have zero height. Is there any way to determine whether a window is rolled up? If so, can we determine its true size and the global coordinates of the top left corner of the content region, so that we can restore and reposition the window when the document is reloaded from disk?*

A When WindowShade "rolls up" a window, it hides the content region of the window. You can tell a window is rolled up when its content region is set to an empty region and its structure region is modified to equal the new "shaded" window outline. WindowShade doesn't do anything with the graphics port, though, so if you need to store the window's dimensions before closing it, use the window's portRect.

With regard to the window's position, WindowShade modifies the bottom coordinates of the structure and content regions of the window, but the top, left, and right coordinates are not changed. These are global coordinates, so you can use the top and left coordinates to track and save the global position of the window on the screen regardless of whether the window is rolled up.

Q *Sometimes balloons won't show up when I call HMShowBalloon; I get a paramErr (-50) instead. The hmmHelpType is khmmTEHandle. HMShowBalloon calls TextWidth on the hText of my TEHandle (the result of which is 1511, the width of 338 characters), then multiplies that by the lineHeight (12), yielding 18132. It then compares this to 17000, doesn't like the result, puts -50 into a register, and backs out of everything it has done previously. What's the Help Manager doing?*

A The Help Manager checks against 17000 to ensure that the Balloon Help window will always be smaller than a previously determined maximum size. Currently, you're limited to roughly the same number of characters with a styled TEHandle as you are with a Pascal string: 255 characters.

Keep your help messages small by using clear, concise phrases. If you absolutely need more text in a balloon, you can create a picture of it and use khmmPict or khmmPictHandle to specify it for your help message. This is not recommended, however; "picture text" has the disadvantage of being difficult to edit or translate to other languages.

Q *Is there any way I can stop LClick from highlighting more cells when the user drags the cursor outside the list's rView area? My program allows users to select more than one*

item from a list and then drag and drop these selected items into another list. But I run into a problem with the LClick function: when I drag these items outside the list's rView area, it still highlights other cells. What can I do?

A If you want to use LClick and not change the highlighting of cells when the cursor leaves the rView of the list, you should install a click-loop procedure that tracks the mouse. When the mouse is outside your list's rectangle, return false to tell the List Manager that the current click should be aborted. It turns out that this is a nice way to start a drag as well, since you know that the mouse has left the rectangle. It might look like this:

```
GetMouse(&localPt);
if (PtInRect(localPt, &(*list)->rView) == false)
   return false;     // We're out of the list.
else
   return true;
```

Q *I'm developing a Color QuickDraw printer driver and want to match colors using ColorSync 1.0.5 with a custom CMM. I was told that for efficiency I should manually match colors inside my Stdxxx bottlenecks, instead of calling DrawMatchedPicture. Is this really more efficient? Why?*

A Surprising as it may be, it is more efficient for printer drivers to manually match colors inside Std*xxx* bottlenecks than to call DrawMatchedPicture. This is because ColorSync 1.0's DrawMatchedPicture doesn't use bottlenecks as you expected. It does install a bottleneck routine that intercepts picture comments (so that it can watch the embedded profiles go by), but it doesn't do the actual matching in bottleneck routines. Instead, it installs a color search procedure in the current GDevice. Inside the search procedure, each color is matched one at a time.

While this implementation has some advantages, it's painfully slow on PixMaps, because even if the PixMap contains only 16 colors, each pixel is matched individually. This has been changed in ColorSync 2.0. To boost performance, PixMaps (which are, after all, quite common) are now matched in the bottlenecks instead of with a color search procedure. (See the Print Hints column in this issue of *develop* for more on ColorSync 2.0.)

Q *I need to add some PostScript comments to the beginning of the PostScript files generated by the LaserWriter GX driver. On page 4-119 of Inside Macintosh: QuickDraw GX Printing Extensions and Drivers, it says that you can override the GXPostScriptDoDocumentHeader message to do this. I wrote a QuickDraw GX printing extension to implement this, assuming that all I had to do was to override the GXPostScriptDoDocumentHeader message and buffer the desired data with Send_GXBufferData. Here's an example of my code:*

```
OSErr NewPostScriptDoDocumentHeader(gxPostScriptImageDataHdl hImageData)
{
   OSErr    theStatus = noErr;
   char     dataBuffer[256];
   long     bufferLen;

   strcpy(dataBuffer, "%%DAVE'S TEST DATA");
   bufferLen = strlen(dataBuffer);
```

```
    theStatus = Send_GXBufferData((Ptr) dataBuffer, bufferLen,
                                                    gxNoBufferOptions);
    if (theStatus != noErr)
       return theStatus;
    theStatus = Forward_GXPostScriptDoDocumentHeader(hImageData);
    return theStatus;
}
```

Unfortunately, this causes a bus error when Send_GXBufferData is called, even if I put Send_GXBufferData after the call to Forward_GXPostScriptDoDocumentHeader. Why doesn't this work?

A The override in your extension is basically correct, but the order of your code needs to be slightly different:

```
// Note that the string is terminated with a return character:
#define kTestStr "%%DAVE'S TEST DATA\n"

OSErr NewPostScriptDoDocumentHeader(gxPostScriptImageDataHdl hImageData)
{
   OSErr    theStatus = noErr;
   char     dataBuffer[256];
   long     bufferLen;

   theStatus = Forward_GXPostScriptDoDocumentHeader(hImageData);
   if (theStatus != noErr)
      return theStatus;

   // Note that we do (sizeof(...) - 1) below to strip off the C string
   // null terminator for the string defined.
   theStatus = Send_GXBufferData(kTestStr, (sizeof(kTestStr) - 1, 0);
   return theStatus;
}
```

Make sure that the string is terminated with a return character. If you're using a **#define** to allocate static space for the string (which is not recommended), remember that it allocates the string plus a null terminator; **sizeof** then returns the size of the string, so you need to subtract 1 from the total. This string should come from a resource or a file.

If you want to add to the header from an application (to avoid writing the extension), you can add an item of type 'post' to the job collection, using the tag gxPrintingTagID. If the first character of this item is a % character, it will appear in the job header.

Q *Our application has multiple QuickDraw GX shapes layered on top of each other. The bottom object is a graphic, and the objects on top of it are text shapes. The text objects are transparent, permitting the underlying graphic to show through. Are there functions in QuickDraw GX to facilitate refreshing the background shapes when characters are deleted in the text layout shapes above it? We need to refresh the graphic with minimal flicker and want to avoid resorting to the standard CopyBits routing.*

A QuickDraw GX doesn't have any direct functions to facilitate refreshing or redrawing only a portion of a shape covered by another shape. However, there are a few methods that can be used in conjunction with various QuickDraw GX

and QuickDraw calls to accomplish your goals. Here are three approaches that might work for you:

- As you know, you can have QuickDraw GX draw directly into a GWorld, and use CopyBits to update the appropriate area. This approach is good if you need to draw QuickDraw and QuickDraw GX objects in the same window.
- If you merge multiple shapes into a QuickDraw GX picture, you can use the picture's clip shape to update the area in question. Make your graphic shape the bottom shape in the picture's hierarchy. This forces QuickDraw GX to draw the graphic as the first shape, with the other shapes drawn on top. QuickDraw GX pictures are smart, in the sense that they respect the clip shapes associated with the picture and all of the shapes contained within the picture.

  To update the smallest possible area, convert the QuickDraw update region to a QuickDraw GX path. Then get the current clip shape of the picture with GXGetShapeClip, and save it for later restoring. Use the path as the "new" clip shape of the picture and draw. Finally, restore the picture's clip shape.
- Create a QuickDraw GX offscreen bitmap to perform flicker-free updating in a manner similar to using CopyBits. This method, though, is based completely on QuickDraw GX. When updating the screen, clip your drawing to the area you want to update. For an example, see the "3 circles - hit testing" sample that ships with QuickDraw GX.

Q *I'm using a layout shape to represent an area for editable text that will have a fixed position, style, font, size, and width. This layout shape has some default text that the user is prompted to change (text content only, no other attributes). Each time text is added (the new text replaces the previous text string), the user interface code checks whether the size of the new string goes beyond the defined width. I do this by comparing the width of the local bounds with the width given within the layout shape geometry. In all cases, the justification setting is 0, but the flush setting varies (left/0, center/0.5, right/1.0).*

Sometimes the width of the local bounds reaches a point where it's wider than the width defined by the shape; in other cases, it approaches the width but never reaches or surpasses it. In this situation, the text is updated and begins to compress itself within the defined width. How can I allow text to be entered till the width is reached, but not compressed?

A The problem you describe was fixed in QuickDraw GX 1.1.1 with a new API call:

```
Fixed GXGetLayoutJustificationGap (gxShape layout);
```

This function returns information that was always generated during layout's justification processing but was never made publicly visible before. It represents the signed difference between the specified width for the layout and the measured (unjustified) width.

By setting a width in the layout options, but leaving the justification factor at 0, you can keep adding text until the results of the GXGetLayoutJustificationGap call changes sign from positive to negative. At that point, the text starts to compress, so you should prohibit new text entry. It's a very fast call (since its

result is cached as part of the layout process anyway), so calling it on every typed character shouldn't slow things down at all.

Some examples may help clarify the use of this call: Suppose you create a layout with the width field of the gxLayoutOptions set to 500 points and the justification factor set to **fract1** (full justification). If the unjustified width of the layout is only 450 points, GXGetLayoutJustificationGap returns +50 points; if the unjustified width is 525 points, this function returns -25 points. A positive value means the line will be typographically stretched to fill the specified width, while a negative value means the line will be typographically condensed.

Note that the justification factor in the gxLayoutOptions doesn't have to be **fract1** in order for this function to return useful results. For instance, if you set a width value but leave the justification factor at 0, the line will not be justified unless its unjustified width exceeds the specified width. In this case, layout will typographically shrink the line. A client program that wants to determine when the end of a line is reached (for line-breaking purposes) can call this function after every character is added (as the user types, for example); as soon as the value becomes negative, the client knows that the margin has been reached.

Q *There are three options on the General panel of the QuickDraw GX Print dialog — Collate Copies, Paper Feed, and Quality — that we would like to move to one of our own panels. We have solutions that differ from the default ones, and we want to rename these solutions and associate them with our printer. How can I eliminate those options from the General panel?*

A There's no mechanism in QuickDraw GX to remove panel items from the standard Print panels, except for the Quality item. The Quality collection item (gxQualityTag = 'qual'), whose structure is defined in PrintingManager.h, has a Boolean field called disableQuality. To eliminate the Quality item from the panel, specify true for the disableQuality field in your driver. Although you cannot remove the other items, you can disable them (dim them in the panel) by getting the collection item and setting the locked attribute with SetCollectionItemInfo.

Q *Do I need to call GXCloneColorProfile before calling GXConvertColor? Since the color passed into GXConvertColor by ColorSync is destroyed, should the color profile passed in as part of the color be disposed of? If not, isn't that a memory leak?*

A Calling GXCloneColorProfile isn't necessary, and it would require additional work that doesn't need to be done. The gxColor structure is a public data structure, not an object: the application, not QuickDraw GX, handles adding and maintaining references to objects with respect to gxColors (and gxBitmaps). QuickDraw GX maintains owner counts when the profile is attached to another QuickDraw GX object (using GXNewBitmap, GXSetInkColor, and so on). This is not a memory leak.

For example, consider this scenario: When an application gets a shape's color, the ink's profile has two owners — the shape and the application. Therefore, the application can reference the profile in gxColor structures, even if the shape is disposed of. Once the application calls GXDisposeColorProfile, the reference is no longer valid. Cloning the color profile does nothing except to require that GXDisposeColorProfile be called afterward. As a result, all that happens is that time is wasted as the owner count goes from a positive number to that number plus 1, and then back down.

Q *Does QuickDraw GX send the GXDoesPaperFit message when I'm setting up input tray dialogs, or is the driver supposed to do this? If QuickDraw GX doesn't, it's possible for users to request completely invalid paper sizes, which can violently crash most raster drivers.*

A QuickDraw GX sends the GXDoesPaperFit message in the default implementation of the input trays dialog to constrain the configuration options, and drivers that perform their own input trays dialog should do the same. A driver should override this message if it needs other than the default logic, which responds that everything fits.

The packing buffer size specified in the 'rpck' resource is set to the expected maximum size needed. Unfortunately, this is far smaller than what's needed when handling larger than expected paper sizes. To work around this, you can set the packing buffer size so that it can accommodate the largest paper size the printer can use.

Q *I've been experimenting to see what happens when a print job is canceled part of the way through. If I cancel when GXOpenConnection and GXStartSendPage have both completed successfully, I get unexpected GXCleanupOpenConnection and GXCleanupStartSendPage messages. If I cancel at another point in the job (for example, during rendering via the Remove button in the desktop printer status window), GXCleanupStartSendPage and GXCleanupOpenConnection messages are passed through after ImageDocument exits. This behavior seems very odd, and it doesn't appear to be discussed anywhere in the documentation. Shouldn't GXCleanupOpenConnection and GXCleanupStartSendPage be called only if their respective routines return an error?*

A The unexpected GXCleanupOpenConnection and GXCleanupStartSendPage messages are coming from the default implementations of ImageJob and ImagePage. The ImageJob code calls Send_GXSetupImageData, and if an error occurs, it sends GXCleanupOpenConnection. ImagePage calls Send_GXRenderPage and sends GXCleanupStartSendPage if an error occurs. If GXStartSendPage or GXOpenConnection doesn't complete successfully, the respective cleanup calls are not sent. Although the documentation states otherwise, this behavior is correct.

Q *Is the layout of the PostScript printer preferences ('pdip') resource documented correctly in Inside Macintosh: QuickDraw GX Printing Extensions and Drivers?*

A No. There's a bug in the documentation for the 'pdip' resource on page 6-88 of *Inside Macintosh: QuickDraw GX Printing Extensions and Drivers*. The render options field is in fact a long word. The resource is defined correctly in the interfaces (PrintingRezTypes.r) and in the MPW 411 files for QuickDraw GX.

Q *Is there a simple way to detach a QuickTime movie from its original file? I'm trying to place a copy of a QuickTime movie in my application's resource fork.*

A See John Wang's QuickTime column in *develop* Issue 17. You can use the technique presented in that column to extract a movie and put it into the resource fork of a different file. You'll find the column and the accompanying sample code, MultipleMovies, on this issue's CD.

Q *How can I determine whether QuickTime 2.0's MIDI music function is available and whether the larger set of 41 instruments is available? If the MIDI function is available, we need to add code to enable the music portion of our game.*

A The QuickTime Music Architecture became available in QuickTime 2.0 (as described in David Van Brink's article in this issue of *develop*), so checking the QuickTime version in a Gestalt call (selector gestaltQuickTimeVersion) will tell you if the MIDI function is present.

When the QuickTime Musical Instruments Extension is installed in your System Folder, it gives you the musical instruments supported by Apple. This extension is actually a component. If you need to know whether the instruments are present, call FindNextComponent, searching for a component that has a type of 'inst' and a subtype of 'ss '. Here's a code snippet:

```
pascal Boolean AreQuickTimeMusicInstrumentsPresent(void)
{
   ComponentDescription    aCD;

   aCD.componentType = 'inst';
   aCD.componentSubType = 'ss  ';
   aCD.componentManufacturer = 'appl';

   if (FindNextComponent((Component)0, &aCD) != NULL)
      return true;
   else
      return false;
}
```

Q *Are there any known compatibility problems between QuickTime 2.0 and QuickTime for Windows? I'm creating a dual-platform application and want to use QuickTime 2.0 for the video. Is there anything that I should avoid on either platform, or anything I should watch out for?*

A In most cases, you don't have to be concerned about using the same movie for playback on both platforms, as long as the movie is in a flattened format and in a single-fork file. To be sure your movie files are single-fork files, select "Make cross-platform" in the MoviePlayer application when saving your movies (or do something analogous in other applications that produce cross-platform movies).

QuickTime supports sound, video, text, music (MIDI), and MPEG tracks under both Windows and the Mac OS. One difference between the two versions is that you can have only one of each track open under Windows (except for the number of sound tracks; starting with QuickTime for Windows 2.0.1, you can enable/disable multiple sound tracks).

The biggest difference between the two versions is the API: QuickTime for Windows 2.0 doesn't support all the API calls available under the Mac OS. Nearly all of the movie controller APIs are supported, as well as many of the basic calls, but the calls to create manipulable movie tracks are missing. You can't create specific media handlers with QuickTime for Windows 2.0, but you can write data handlers and codecs for the Windows environment.

While working with QuickTime for Windows, you'll have to keep track of all the possible configuration issues that users might encounter. We distribute

README files with the latest information about compatibility and configurations (video/sound cards, drivers, and so on).

For additional information, the Mac OS Software Developer's Kit includes detailed documentation regarding API and architecture issues concerning QuickTime and QuickTime for Windows. Also see *How to Digitize Video* by Weiskamp and Johnson (Wiley Press) for another good source of information regarding the practical issues of both QuickTime and AVI movie creation. Although this book is a bit out of date in the details (it was written to cover QuickTime 1.6.1 and QuickTime for Windows 1.1.1), much of it is still valid.

Q *Can we use a different A5 world with QuickTime? Our plug-in architecture uses A5 for global access, but we allow the A5 world to move. QuickTime doesn't seem to be able to deal with this, and it doesn't realize that EnterMovies was called once the A5 world moves. We currently work around this by locking down our A5 world, but we would rather not do this. If we need to keep doing this, is locking down the A5 world an adequate fix, and can you recommend another solution?*

A You can use a different A5 world with QuickTime, but whatever A5 world you use, you'll have to lock it down. QuickTime allocates a new set of state variables for each A5 world that's active when EnterMovies is called. However, since QuickTime uses A5 to identify each QuickTime client, if A5 changes (your A5 world moves), QuickTime won't recognize that you've called EnterMovies for that client.

Q *How do I determine the correct time values to pass to GetMoviePict to get all the sequential frames of a QuickTime movie?*

A The best way to determine the correct time to pass to get movie frames is to call the GetMovieNextInterestingTime routine repeatedly. The first time you call GetMovieNextInterestingTime, its flags parameter should have a value of nextTimeMediaSample + nextTimeEdgeOK to get the first frame. For subsequent calls, the value of flags should be nextTimeMediaSample, and the whichMediaTypes parameter should be VisualMediaCharacteristic ('eyes') to include only tracks with visual information.

Q *I noticed that certain commercial candies have no taste when they initially hit my tongue. It's only after I start sucking them that the flavor appears. I think there's some sort of coating on them. What is it? Is it harmful?*

Also, what is it that creates that beautiful high gloss I get with my car wax and floor wax? I just love the way it shines after a good hard buffing.

A Carnauba wax is the answer, in both cases. It's a hard wax obtained from the leaves of a Brazilian palm tree (*Copernicia prunifera*), and is used a lot in polishes of all types. It really does buff up beautifully, doesn't it? It also is completely tasteless and nontoxic, and makes a dandy confectioner's glaze, used to keep the candy from sticking to itself.

These answers are supplied by the technical gurus in Apple's Developer Support Center.•

Have more questions? See the new Macintosh Technical Q&As on this issue's CD. (Older Q&As can be found in the Macintosh Q&A Technical Notes on the CD.)•

DAVE JOHNSON

THE VETERAN NEOPHYTE

A Feel for the Thing

I used to think there was no room for mystery in the world of computers. I didn't think there was any use for fudge factors or rules of thumb or hunches in the clean, exact, hermetically sealed bubble of logic we all spend so much time diddling and poking. That stuff belongs to "real world" engineering, not software engineering, right? Software is always bounded and orderly, always understood completely from top to bottom, with no dangling ends, no frayed edges, and no baling wire and duct tape holding things together. There's never a need for vague, hand-waving explanations of how it all works, because we know how it works.

That's what I used to think. I'm not so sure anymore.

Ultimately, of course, the operation of computers is deterministic and absolutely predictable. There's guaranteed to be a complete explanation for any event on the computer; the search for an answer will always find one. It's like playing Go Fish with a deck of cards that contains only threes — "Got any threes?" "Yep." "Got any threes?" "Yep." "Got any threes?" "Yep." The answer itself, of course, may be convoluted and difficult, and is often *way* too much trouble to actually track down ("Have you tried rebooting?"), but it's always there. The world inside computers has a definite, impermeable bottom, like a swimming pool.

The real world, on the other hand, is more like being out in the middle of the ocean: the bottom is nowhere in sight, and in fact is so far away that it may as well not exist at all. Trying to completely explain things in the real world is generally an exercise in futility, though one that humans seem to have a capacious appetite for (that's what science is all about, after all). The real world is so vast and complex that our explanations are never really complete. The answers always lead to more questions, and the edges of our knowledge remain frayed and ragged and crumbling, even though the center may have a seemingly solid, well supported integrity.

The thing that got me thinking about all this is boomerangs. I've been learning to throw boomerangs lately, and it's extremely satisfying — and somehow endlessly novel — to throw something away from yourself as hard as you can, and have it return several seconds later, hovering gently down into your waiting hands like a bird coming home to roost. (Such a perfect flight, of course, is a rare thing for a novice like me. More often, if the boomerang comes anywhere near me, it's slicing past at a frightening rate of speed while I cringe, covering my head.) While I've been learning to throw boomerangs, I've also been trying to watch myself learn to throw boomerangs — sort of meta-boomeranging — and I noticed that a complete explanation of what was happening was not only absent, but completely unnecessary: I don't need to know how boomerangs work to learn to throw them well.

Boomerang throwing is one of those real-world activities — there are many of them — that are governed by rules of thumb, by approximation and estimation, and by "feel." There are lots of variables involved in producing a good boomerang flight, and they're all sort of woven together, interconnected and interdependent. The direction of the throw, the angle of the boomerang as it leaves your hand, the forward power of the throw, and the amount of spin all contribute to the flight characteristics, but the way they combine and interact is complex and nonobvious. How's a poor, bewildered boomerang neophyte to make any sense of it all?

Well, the only way to learn to throw boomerangs is to get yourself a decent boomerang (very important!), read a little about it or get a lesson from someone, and then just get out there and start throwing. You need to experience it; you need to feel the smooth, flat weight of the thing, notice the way it slices the wind as it leaves your hand, and watch as it spins and swoops. Every throw you make adds to a growing store of knowledge about boomerang behavior. Slowly, you begin to sense the structure of the rules that govern the flight of the boomerang, to get a feel for it, to gain some control. But no matter how long you work at it, there's always more you can learn about boomerangs. Boomerang throwing, like most things in the real world, has no bottom.

But even though things in the real world are webby, tangled, and complex, with no real bottom and no real

---

DAVE JOHNSON has an ever-lengthening list of life goals, things that he'd like to accomplish or experience before leaving this mortal coil. Some recent additions include making marshmallows from scratch, milking a cow, and hugging a full-grown bear. (Is bear breath better than dog breath? There's only one way to find out!) If you have a cow or bear Dave could visit, please let him know.

center, and even though complete understanding is out of our reach, that doesn't stop us from getting things done. Even though we may not understand exactly what's going on when we throw a boomerang, we can learn to throw them anyway, and can actually learn to throw them with incredible skill. Scientists don't have a complete understanding of fluid mechanics, but we can still design hydraulic lifts that lift, toilets that flush, and airplanes that fly.

Though it seemed profound when I first thought of it that way, it really isn't anything remarkable at all. It's the stuff our everyday sensory world is made of. It's our standard, animal mode of operation. We depend heavily on trial and error, on finding and keeping strategies that work. We invent myths and superstitions to explain things we don't understand, we guess, we fake it, we operate by feel. And it works just fine.

But we don't need that sort of thing in the clean, deterministic world of computers, right? If we know the answer is within our reach, then why gloss over it? There's one very good reason: it's gotten to the point where it's often really hard to reach the answer. Computers have become so complex that finding the real answer is often a Herculean feat requiring great effort and stamina. The things that we're "growing" in the machine are getting very deep and webby and complex, just like things in the real world. That nice smooth bottom we all know and love is getting pretty remote and hard to see, and in fact trying to keep it in sight often holds us back.

The truth is, we *need* fakery, or myth, or something similar, to avoid being hopelessly mired in complexity, and to let us feel cozy even in the face of something too deep to comfortably understand. The idea that an icon in the Finder, a document window in an application, and a file on the hard disk are all "the same thing" is a fiction, an illusion created from smoke and mirrors, and one that users don't even think about anymore (unless, of course, an application screws up the illusion; see Mark Linton's article in this issue for some code to help you avoid such a faux pas). But it's precisely that kind of myth and abstraction that lets people ignore all the underlying complexity and just go about their everyday business. Without that kind of trickery most people would be lost.

Humans have a deep need for some sort of explanation, and we'll often ignore aspects of a situation, or even make stuff up out of thin air, if it helps us to find an "answer." Remember the frictionless inclined planes and perfect vacuums of college physics? Without that kind of glossing over of details, we'd have been helpless. (A college housemate of mine and I used to joke about running a college physics stockroom: boxes of frictionless, massless pulleys on the shelves; gallon jugs of zero-viscosity liquid at our feet; coils of infinite and semi-infinite wires hanging neatly on the pegboard wall. Those wires have no thickness or mass, thank goodness, or the storage requirements would be prohibitive.) This need for explanation is what has led us to science, and to religion, and to superstition. These are not the same thing, of course, but they can all serve the same purpose: a soothing, protective balm on the raw edges of our incomplete knowledge. They give us a ground to stand on, a rail to hold on to, as we totter along in the darkness, going who knows where, hoping the batteries will hold out long enough to get an answer.

Now that I think about it, I'm happiest with a generous helping of myth and fiction stirred into my computing. It can help make the computer — which, let's face it, is essentially a gritty, sharp-edged, and hostile machine — feel more rounded and friendly. It can provide a useful disguise, like a plastic nose and glasses on something seething and alien, making it recognizable, familiar, even comforting and amusing. If it's done well, it can even let me learn to use a computer in much the same way I learn to throw a boomerang: by picking it up and trying it, by mucking around and getting a feel for it, by discovery.

Maybe best of all, it lets computers keep a little of their mystery. The mystery and magic of the Macintosh are why many of us are programmers, after all. Mysterious things, things that don't have clean and obvious boundaries, are inevitably more interesting and more fun. There's no denying that computers have a dull, featureless, dreary bottom. But in the other direction there seems to be no boundary; the top, if there is one, is as far away as the sky. So yes, I think there's plenty of room for mystery in the world of computing. Plenty of room indeed.

RECOMMENDED READING

- *Many Happy Returns: The Art and Sport of Boomeranging* by Benjamin Ruhe (Viking, 1977).
- *How to Hide Almost Anything* by David Krotz (William Morrow and Company, 1975).

Thanks to Lorraine Anderson, Jeff Barbose, Brian Hamlin, Mark "The Red" Harlan, Bo3b Johnson, Lisa Jongewaard, and Ned van Alstyne for their always enlightening review comments.

Dave welcomes feedback on his musings. He can be reached at JOHNSON.DK on AppleLink, dkj@apple.com on the Internet, or 75300,715 on CompuServe.

Newton Q & A: Ask the Llama

Q *I have a program that communicates with the desktop. Part of the information sent is real numbers. I've found functions to stuff almost every other type of data into a binary object except real numbers. How do I do that?*

A You have two choices. First, you could just print the real number as a string (using SPrintObject), send the string, and convert it back on the other side. Clearly this isn't a good idea if you want to maintain a high degree of precision. The other choice is to construct the correct type of binary object for the target desktop machine. In other words, take the Newton real representation and convert it into, say, IEEE floating point. Then you can use BinaryMunger to stuff the binary object into whatever packet of data you're constructing.

Note that Newton uses SANE representation for real numbers that are in the representable range. However, the representation of exceptions (such as NAN and infinity) are different and undocumented. At this time you should avoid converting these types of real numbers.

Q *Can you give me a short and clear description of the different types of Newton memory?*

A There are three important "pools" of so-called internal memory, each with different tradeoffs.

The NewtonScript heap (about 90K to 96K on current devices) is where all the runtime data from NewtonScript lives. Any result from the Clone family of calls will take up NewtonScript heap space. The view frame made at run time from your application templates will take up this heap space. NewtonScript heap space is very precious, so you should try to use as little of it as possible, especially when your application's base view isn't open.

The user store (192K in the MessagePad 100, larger on other devices) is where application packages stored internally live, and where soups are located. The entries in the soups are located in this space. While not quite as precious as the NewtonScript heap, this space can certainly run out. This is the space that's "extended" when a RAM PCMCIA card is inserted.

There is also some system heap space, which is used for, well, everything else. The viewCObjects and drawing objects live here. Recognition uses memory from here. You can run out of this space (in which case you get the Cancel/Restart dialog) but it's less of a programming issue.

Q *I have an application that uses a protoRollBrowser. When I expand the items, they have lines separating them. I can't seem to get rid of them. Is this a bug?*

A What you're seeing is part of the default definition of a protoRollItem. It includes a 1-pixel border. You can remove that border by modifying the viewFormat of your rollItems. In addition, you may want to set the fill to white.

Q *I'm using a protoRoll (not protoRollBrowser) in my application. But it never shows up. What's the problem?*

The llama is the unofficial mascot of Developer Technical Support in Apple's Newton Systems Group. Send your Newton-related questions to NewtonMail DRLLAMA or AppleLink DR.LLAMA. The first time we use a question from you, we'll send you a T-shirt.•

A You need to give it a viewFlags slot and make sure the Visible bit is checked. The default is Application and Clipping, but this won't make the protoRoll visible if it's included inside another view.

Q *I have a text view that the user can use to enter text. I wanted to extend a selection. I knew the insertion caret was at the end of the selection, so I called SetHilite(newPoint, newPoint, nil), where newPoint is the new position for the selection extension, but I got no highlight. What's going wrong?*

A The behavior is actually perfectly correct. There's a not quite obvious interaction between the caret and SetHilite. As shown in the table below, how SetHilite behaves depends on four things: the **start** and **end** character positions (the first two arguments) being equal, the value of **unique** (the third argument), the presence of a previously highlighted selection, and the presence of the caret. Note that the following explanation refers to the case of a single paragraph view, in which there can be only one selection; if there are multiple paragraph views, it's possible (with **unique** nil) to have multiple discontiguous selections.

Highlight and unique	**When start = end**	**When start <> end**
No previous highlight, **unique** true or nil	If there's a caret, move caret; otherwise, no effect	Create new highlight from **start** to **end**
Previous highlight, **unique** true	Clear highlight and, if there's a caret, move caret	Create new highlight from **start** to **end** (remove old highlight)
Previous highlight, **unique** nil	Extend highlight to include **start/end**	Extend highlight to include **start/end**

Q *I have an application that uses ADSP to connect to a server on the desktop. I want the server to handle multiple Newtons connected simultaneously. Unfortunately, if a connection fails after it's opened, the server doesn't seem to be able to identify it as a new connection when the Newton reconnects. This causes problems in the server's ability to handle multiple connections. Can you help?*

A We'll assume that the Newton tries to reconnect shortly after losing the connection. In that case, the Newton doesn't generate a new connection ID, so your server probably acts as if the connection didn't close, while the Newton is acting as if it's establishing a new connection. Currently the only solution is to force the Newton to wait three minutes after an improper disconnect before trying to reconnect.

Q *I have a communications program that always sends a string of the same size to the desktop. The string is quite large, and I would like to preallocate it and fill it with a particular value. What's the best way to do this?*

A As with all things in programming, the answer is a tradeoff between space and time. Let's assume that you want a string of 2K characters filled with the character A, and that you control the contents of the string (that is, if you get user input, you make sure the input is a string). The first option is to allocate the string at compile time. Note that you shouldn't allocate your string constant with a double-quoted string ("a string"), since typing 2K (less the terminator) characters is monotonous and error prone. The way to allocate the string is with the following SetLength trick:

```
constant kNumberOfUnicodeCharsForString := 2048;  // 2K chars
DefConst('kMyBigString, call func()
   begin
      // SetLength uses bytes; Unicode chars are 2 bytes each
      local aStr := SetLength("",
            2 * kNumberOfUnicodeCharsForString + 2);
      // initialize the string
      for i := 0 to k1KUnicodeChars - 1 do
         aStr[i] := $A;
      return aStr;
   end with ());
```

At run time you can clone kMyBigString and do what you need to fill it with characters. Note that the object is not a string; you would need to use StuffByte to put in individual characters.

The advantage of this method is that it's very fast: it averages less than one tick (60th of a second) for the clone. The disadvantage is that it puts a 4K object in your package (Unicode strings are two bytes per character). If you can't afford the 4K in your package, you need to generate the string at run time. Using the above code at run time averages 52 ticks.

Another possible runtime method is to use smart strings, which allow you to preallocate strings and concatenate them in a more efficient way. The first attempt at doing this seems to be inefficient, at an average of 175 ticks:

```
// defined constant somewhere in your project
constant  kNumberOfUnicodeCharsForString := 2048;

local s := SmartStart(2 * kNumberOfUnicodeCharsForString + 2);
local l := 0;
for i := 1 to kNumberOfUnicodeCharsForString do
   l := SmartConcat(s, l, "A");
SmartStop(s, l);
```

However, simply concatenating two characters at a time reduces the average to 88 ticks; four characters reduces it to 44; and so on. A lesson here is that testing and measurement are your friends.

Q *I'd like to train my dog to code in NewtonScript. How can I do that?*

A I'm afraid the prospect isn't promising. Dr. J. L. Fredericks at SITAP (Stanford Institute for Training Animal Programmers) has been trying for ten years to train different animal species to program computers. Although he's had some success training dogs to do simple programs, he says, "Anything more than a simple statement is beyond them. No loops, no conditionals." Besides which, paws don't work well for moving mice. For Newton programming the best he has been able to achieve is training a rat to reset the Newton on command. As Dr. Fredericks says, "Never underestimate the usefulness of a ratset."

Thanks to our Newton Partners for the questions used in this column, and to jXopher Bell, Bob Ebert, David Fedor, Neil Rhodes, Jim Schram, Maurice Sharp, and Bruce Thompson for the answers. Thanks especially to Bob Ebert for the Newton memory description.•

Have more questions? Take a look at Newton Developer Info on AppleLink.•

Video Nightmare

See if you can solve this programming puzzle, presented in the form of a dialog between a pseudo KON (Ian Hendry) and BAL (Eric Anderson). The dialog gives clues to help you. Keep guessing until you're done; your score is the number to the left of the clue that gave you the correct answer. Even if you never run into the particular problems being solved here, you'll learn some valuable debugging techniques that will help you solve your own programming conundrums. And you'll also learn interesting Macintosh trivia.

IAN HENDRY AND
ERIC ANDERSON

BAL I've got one for you, KON: I just updated to System 7.5 on my 8100/80 AV. Everything seemed OK for a while. I was comparing Scenery Animator to Vistapro and I noticed that my cool new desktop pattern had disappeared. It was there when I booted, but just as the Finder was coming up, the desktop changed to a black-and-white pattern.

KON That's easy. Go back to System 7.1 and the world will be fine again. Next.

BAL Hey, 7.5 is the source of much wonderment. It's really a lot of fun!

KON I don't know that much about 7.5. Metrowerks and THINK C seem to run fine on 7.1. Is this part of that new puzzle CDEV that was added to spruce up the system?

BAL Quit trying to change the subject. My desktop pattern went away and I'm not happy about it.

KON Hmm. Did you change anything on your machine?

BAL I turned on VM for the memory-hungry rendering stuff.

KON So turn off VM and see if the problem goes away.

IAN HENDRY (AppleLink HENDRY; Internet hendry@apple.com) gets paid by Apple to work on video stuff. His hobbies include shipping products and collecting new Engineering managers. He can be found skipping meetings to play Ultimate and working all hours to make up for it. Ian's going to be a dad soon, and though he has been in rigorous sleep-deprivation training for years, he's hoping (but still not certain) that he's ready for what he's gotten himself into.

ERIC ANDERSON (AppleLink ERIC3) skipped out on the OS Services group at Apple to get away from the chore of working on VM and the Thread Manager. Now he works as Ian's evil twin on video-related Mac OS issues — and he gets bugs assigned to him that state, "When using a multisync monitor with my threaded test app while VM is on, this funny thing happens." Seeing that there's no escape, Eric wants more than ever to move to Hawaii and build boats.

BAL Wow, that worked great. Now all I have to do is buy $1800 worth of tariff-enhanced RAM so I can render my flyby of the Pentagon.

KON I won't ask what you're up to these days. My recent stock dealings have left me low on bail money. Did you try it on 7.1.2?

100 BAL It didn't happen on straight 7.1.2, but I installed System Update 3.0 and it happened. VM must have changed in this update.

KON Sounds like a VM problem all right. Paste the older VM resources into the new system. (I love component software.)

BAL The problem is still there. Does this let VM off the hook?

KON VM is never off the hook, but if your only problem is that the desktop pattern is black and white, maybe you should stop whining and do your work.

95 BAL No, this is more serious. I opened the Monitors control panel and there was no depth list and no monitor tile in the rearrange section.

KON Well, this should be pretty straightforward. Does it happen every time?

90 BAL No such luck. This one is really evil. I've been trying to get a reproducible case for days. Sometimes it happens right away, sometimes it goes away for hours. Once it starts happening, it seems to keep happening across restarts. It doesn't happen as reliably across shutdowns. It seems to happen more in millions of colors but will happen at other depths too. Switching the display back and forth between a 13-inch and a multiple-scan display sometimes causes the problem to show up. Changing VM and RAM disk settings seems to affect the reproducibility.

KON Cool! The random bugs are always the most fun. Let's get our trusty MacsBug and see if we can find where it's going bad. Look at the video driver and GDevice.

85 BAL When I try to enter MacsBug, the mouse freezes but MacsBug doesn't come up.

KON Dang, I hate it when the tools don't work.

BAL Perhaps we should devote this column to model rocketry and video games instead.

KON Now there's a thought. If I type Command-G, does the mouse unfreeze, or do I have to reboot?

BAL The machine comes back when you hit Command-G.

KON So MacsBug is there; you just can't see it. I'll use **log foo** to dump the output to a file named **foo**, followed by

```
dm @@thegdevice gdevice
```

Then I'll use **drvr** to see if the video driver is alive.

BAL Nice **log** trick, KON! The GDevice is fine, and the driver looks as if it's loaded and active.

KON Drivers and VM sometimes don't get along. Maybe the driver is doing something wrong. Did you try other video cards?

80 BAL It seems to happen only on Power Macintosh AV models.

KON It's the nasty "*fung*us" problem from Issue 17 all over again, or maybe your card has gone bad.

BAL Well, I'm pretty sure there was no "*fung*us" around when the AV card was developed, so it's probably not that. Besides, we're sticking to production software in this column from now on. Anyway, I thought it might be my card, too, so I borrowed your card. Thanks for the loaner. I turned on VM and it worked like a champ . . . for a while. Now I've got the same problem again.

KON You killed my card?

BAL Admittedly, it's a computer that can do anything. But for the purposes of this column, that idea is pretty far-fetched.

KON Here's what might be happening: Early in the boot process, VM isn't present. For each card, the system calls the card's PrimaryInit code and creates a GDevice. When VM loads, it changes the logical address mappings. When the driver is called again, it assumes a one-to-one logical-to-physical mapping of RAM, so the card starts responding to bogus address cycles. This confuses the card's bus translator, and . . .

75 BAL Whatever. Any other stabs in the dark? Lose five points and try again.

KON OK. Perhaps there are subtle timing variations when VM is present, and the video card might have borderline hardware that's affected by these timing dependencies. Or maybe the card's controller gets into a state where it no longer responds to its address space.

70 BAL You're getting desperate. It's not a hardware problem. The declaration ROM is there and everything looks fine. You can't blame this on the hardware. Let's once again follow the software decision tree.

KON Yeah, you're probably right. Now that I think about it, those ideas seem really out there.

So what you're telling me is that the desktop pattern is black and white, MacsBug isn't working, and the Monitors control panel doesn't show depths or a monitor tile. Let's find out when MacsBug stops working.

65 BAL When the machine boots, MacsBug is working, but by the time you get to the Finder, it's gone. It seems to vanish early in the boot process.

KON See if you get to the first Display Manager call with an **atb DisplayDispatch** or **atb ABEB**.

BAL OK. MacsBug is still alive.

KON I'll set a breakpoint just after the first Display Manager call and then **go**.

BAL Yep. There doesn't seem to be a problem now. But the weird thing is, if you trace over the Display Manager call and then type **go**, MacsBug will eventually go away.

KON Wacky. Sounds like a Display Manager bug.

BAL Earlier you said it was a VM bug.

KON Both have been convicted criminals in the past, so you can't blame me for thinking they're suspects. I'll bet you a buck it turns out to be neither! Do an **atb** on the Display Manager call and trace from there until MacsBug goes away; it shouldn't take too long.

60 BAL Sorry, MacsBug never goes away. The problem isn't reproduced.

KON So what you're telling me is that if I trace over the Display Manager call and then **go**, I can't get into MacsBug after I'm done booting. But if I keep tracing, the machine boots fine and MacsBug is always available.

BAL That's right. Let me help you along a little bit. There's an SSecondaryInit call (which runs SecondaryInit code for the video cards) just a few 680x0 instructions after the first Display Manager call. Does that help at all?

KON What happens if we do an **atb** on SSecondaryInit?

55 BAL I can't reproduce the problem. If I set a breakpoint just after the Display Manager call and **go**, the problem disappears. If I do an **atb** on the Display Manager call, and either **go** from there or trace over it and **go**, the problem happens. If I trace over it for a few instructions and **go**, the problem doesn't happen.

KON So, what are the "few" instructions? It looks like they're the ones whacking the video driver.

50 BAL No, they're just a few MOVE instructions to innocuous RAM locations — nothing that should touch video.

KON What does this Display Manager call do? Could it be hosing anything in the slots?

45 BAL I don't think so. I used MacsBug to skip it entirely and the problem still happens.

KON This isn't getting us anywhere. Maybe the desktop pattern problem has some better clues. The General Controls control panel in System 7.5 has an INIT resource that calls HasDepth to decide whether to use the color pattern. It then sets a PRAM bit to remember whether to use a color pattern across restarts. When the desktop pattern is black and white, I'll use the **log** trick to find out what the HasDepth call is returning.

40 BAL It returns an error.

KON Aha! Since HasDepth returns an error, the INIT resource thinks it's on a display that can support only one bit per pixel (black and white), so it disables the color desktop pattern and resets the PRAM bit; the color desktop pattern is now gone forever.

BAL OK.

KON Let's trace HasDepth and find out what's wrong.

35 BAL It looks as if the Slot Manager returns the correct values for the active functional sResource of the card but fails to find the depth. It returns -316, an smInitStatVErr. According to Errors.h, this error indicates that the siInitStatusV field was "negative after primary or secondary init." This means the card's PrimaryInit or SecondaryInit code returned an error.

KON We can bet it's not PrimaryInit, because the GDevice is good. If the error had happened in PrimaryInit, QuickDraw would have gotten an smInitStatVErr when it called the Slot Manager to build the GDevice.

30 BAL You're finally making some progress!

KON MacsBug also makes Slot Manager calls (when it tries to switch depths), which explains why it fails. That means the problem must be with the SSecondaryInit call. Once the Slot Manager gets this error, most Slot Manager calls return errors.

BAL But this doesn't explain what's causing the Slot Manager to fail to begin with, or why the problem goes away every time we get close to it with MacsBug.

KON Maybe we should try this with a BootBug card. Can you still get one?

BAL Maybe, but we're doing pretty well here. Let's keep going a little longer.

KON Let's try to figure it out by brute logic. What does the SecondaryInit code look like on this card?

25 BAL MOVE.L A0,A0.

KON That's it? Two bytes? No RTS? Cool! A bug in the AV card ROM! Does this mean we all get new cards with the new 2.0 ROM? Maybe they can simplify that complicated Monitors control panel Options dialog at the same time. How does it boot at all?

20 BAL Good question. *Designing Cards and Drivers for the Macintosh Family* says that the SecondaryInit entry on a video card is an SExecBlock, which is a header followed by actual code. The Slot Manager validates the header before it executes the code. The first byte of an SExecBlock is the revision number, and the Slot Manager checks for a revision byte of 0x02. Since MOVE.L A0,A0 is 0x2048 in hex, the first byte of the AV card's SecondaryInit entry is 0x20, which is a bogus entry, and the Slot Manager will never try to execute the SecondaryInit code.

KON So it's pretty lame, but it should work, right?

BAL Yes. Remember, we added SecondaryInit to the boot process because some machines didn't have 32-bit QuickDraw in ROM. On a machine without 32-bit QuickDraw in ROM, video cards have to disable their functional sResources with direct bit depths (16 and 32) in their PrimaryInit code, because the PrimaryInit code runs before the disk is up and the cards can't tell if the system has 32-bit QuickDraw installed. SecondaryInit was added to give these cards a chance to reenable those direct depths after 32-bit QuickDraw was loaded from disk. Power Macs obviously have 32-bit QuickDraw in ROM, and this card only runs on a Power Mac, so it doesn't need SecondaryInit.

KON Let's walk through the SSecondaryInit call and see what it does. Why does VM make a difference? And why is MacsBug causing the problem to go away if you set breakpoints?

BAL You're just full of questions, aren't you? You're supposed to be giving the answers!

KON Let's walk through SSecondaryInit.

15 BAL For each card, it looks for a SecondaryInit entry in the card's ROM. The entry contains a header followed by the SecondaryInit code. If there's no SecondaryInit entry on the card, SSecondaryInit bails out early. If there is a SecondaryInit on the card, the Slot Manager tries to execute it with SExec and then checks the status from the SExec call. If the status is negative (an error), the Slot Manager marks the slot with that evil -316 error, and the slot is bad from there on out.

KON So who is responsible for setting the error?

10 BAL The code executed by SExec, in this case the SecondaryInit code, should set the status error. If the header is bad, the code never gets run and the status never gets set.

KON Let me guess: the boot code never initializes the status before calling SExec.

5 BAL Yep. And it's allocated on the stack as a local variable, which means that the status is set to whatever garbage is left on the stack. At this point in the boot process you're still in supervisor mode, so MacsBug is sharing your stack. When MacsBug is used, it pushes stuff onto the stack and then pops it off when it leaves (changing the garbage below the stack in the process). That's why setting breakpoints and tracing mask the problem. BootBug also uses the stack, so it too would have interfered with the bug.

Between the first Display Manager call and the SSecondaryInit, the system allocates stack space for the SPBlock parameter for the SSecondaryInit call. After the SPBlock is allocated, the stack pointer is very close to where the local variables for SSecondaryInit will be allocated. At this point MacsBug's stack usage will affect those never-initialized local variables.

This is something else to add to your list of gotchas for MacsBug: If you're in supervisor mode (as you are at this point in the boot process) and you set breakpoints, MacsBug is sharing your stack, and its use of the stack may affect uninitialized variables. Later in the boot process, VM switches the machine to user mode; from then on, MacsBug and applications use different stacks and MacsBug will not interfere with uninitialized variables on the stack.

KON The garbage that VM leaves on the stack (sometimes) happens to be negative. When the boot code gets to SecondaryInit and allocates variables on the stack, it happens to use an area of the stack affected by VM.

BAL Well, I never turn VM on, so I'm always in supervisor mode, and MacsBug always shares my stack. But now I've finally found a good use for VM: turn it on when I have a bug that's hard to reproduce when MacsBug gets involved, and see if it becomes reproducible.

KON That'll slow your machine down.

BAL Nasty.

KON Yeah.

SCORING

75–100 Excellent; you probably have a video-in jack built right into your head.
50–70 Maybe we should be working for you.
25–45 Maybe you should be working for us.
5–20 Maybe you should stick to television. •

Thanks to Rich Collyer, Kent Miller, Mike Puckett, John Yen, KON (Konstantin Othmer), and BAL (Bruce Leak) for reviewing this column. •

INDEX

N

O

P

Q

R

S

T

U

V

W

X

Z

RESOURCES

Apple provides a wealth of information, products, and services to assist developers. APDA, Apple's source for developer tools, and Apple Developer University are open to anyone who wants access to development tools and instruction. Developers may access additional information and services through Apple's Developer Programs.

APDA offers worldwide access to development tools, resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers periodically receive the *Apple Developer Tools Catalog* featuring hundreds of Apple and third-party development products. There are no membership fees. APDA offers convenient payment and shipping options, including site licensing.

Apple Developer University (DU) provides courses to get you started programming on Apple platforms and Mac OS–compatible hardware, as well as advanced, in-depth training on new technologies such as QuickTime VR, QuickDraw 3D, OpenDoc, Apple Guide, and Newton. In addition to classroom training, multimedia self-paced courses and low-cost mini-course tutorials are available through APDA.

The Macintosh Associates Program is a low-cost self-support program that provides information on new technologies and discounts on equipment. It's the primary program for Macintosh developers who don't need programming-level technical support from Apple.

The Macintosh Associates Plus Program In addition to technical information and discounts on equipment, this program enables Macintosh developers to have up to ten programming-level technical support questions answered (via e-mail) per year. It also includes a subscription to the Mac OS Software Developer's Kit.

The Macintosh Partners Program is for Macintosh developers who need unlimited programming-level technical support (via e-mail). It also includes technology seeding, a subscription to the Mac OS Software Developer's Kit, and more.

The Newton Associates Program is a low-cost self-support program for Newton developers who don't need programming-level technical support from Apple.

The Newton Associates Plus Program enables Newton developers to have up to ten programming-level technical support questions answered (via e-mail) per year.

The Newton Partners Program offers Newton developers unlimited programming-level technical support (via e-mail) as well as hardware purchasing privileges and marketing opportunities.

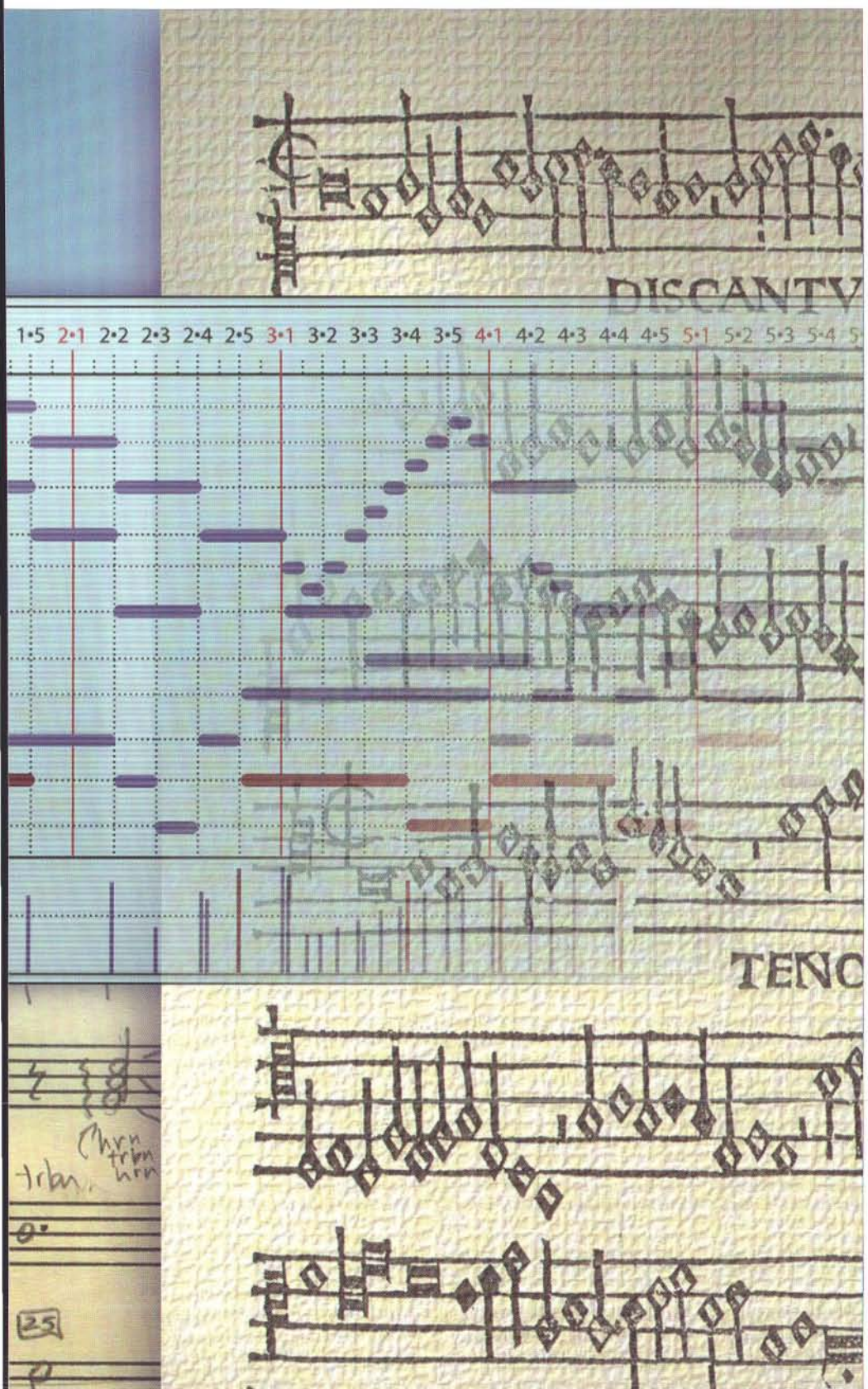
The Apple Multimedia Program is designed for developers interested in the emerging multimedia market. Program features include a quarterly mailing, discounts on third-party products, training, and events.

APDA To order products or receive a complimentary catalog, call 1-800-282-2732 in the U.S., 1-800-637-0029 in Canada, (716)871-6555 internationally, or (716)871-6511 for fax. You can also order electronically (AppleLink APDA; Internet apda@applelink.apple.com; America Online APDAorder; or CompuServe 76666,2405) or write APDA, Apple Computer, Inc., P.O. Box 319, Buffalo, NY 14207-0319.

Apple Developer University (DU) Course descriptions and schedules can be found in the Developer Services areas on AppleLink (Developer Support), eWorld (Apple Customer Center), and the Internet (World Wide Web at http://www.apple.com). Or call (408)974-4897, fax (408)974-0544, AppleLink DEVUNIV, or write to DU at Apple Computer, Inc., 1 Infinite Loop, M/S 305-1TU, Cupertino, CA 95014.

Apple Developer Programs Call the Developer Support Center at (408)974-4897, AppleLink DEVSUPPORT, or write 1 Infinite Loop, M/S 303-2T, Cupertino, CA 95014, for information or an application form. Developers outside the U.S. and Canada should instead contact the Apple office in their country for information about developer programs.

23



Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014